Technical Report

UCAM-CL-TR-905
ISSN 1476-2986

Number 905

Fixed point promotion: taking the induction out of automated induction

William Sonnex

March 2017

This technical report is based on a dissertation submitted September 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

ISSN 1476-2986

Acknowledgements

First and foremost, I thank Lawrence Paulson, my supervisor and mentor throughout my thesis. My gratitude also goes to Thomas Mouat, a true master of the language in which I am supposedly a native speaker, who helped me turn this thesis from an unreadable mess into a readable one. My thanks also go to Alan Mycroft, for his innumerable suggestions and advice, both technical and otherwise.

Sophia Drossopoulou and Susan Eisenbach deserve their own paragraph (this one) for sending me down this theoretical rabbit hole to begin with. Without them I would not have achieved this PhD position and I am eternally grateful for their advice, encouragement, and the opportunities they gave me, both before and during this project.

Another paragraph worthy group of people are Koen Claessen, Moa Johansson, Dan Rosén, and Nick Smallbone. I had a fantastic time visiting their research group in Gothenburg and their enthusiasm for this research field is infectious. I left Sweden with a better understanding of their work and a new perspective through which to view my own.

I met some incorrigible nerds while at Trinity College, some of whom were quite good company. I would like to thank Janina Voigt in particular, for her support and for showing me that object-oriented programming languages might have some merits. Thank you also to James Lloyd for our nerdy cooking sessions, Steffen Lösch for our nerdy gym sessions, and Richard Fletcher for our nerdy cooking and gym sessions.

This work would not have been possible without the financial support of Trinity College Cambridge's external research studentship program, or the incredible group of people I met while I was a student there.

Contents

1 Introduction 9

1.1 Contributions 14

1.2 Thesis outline 15

2 Background 17

2.1 νPCF 17

2.1.1 νPCF syntax 18

2.1.2 Typing νPCF terms 20

2.1.3 Operational semantics of νPCF 22

2.1.4 Syntactic sugar for νPCF 23

2.1.5 Fold 25

2.2 Domain Theory 27

2.2.1 Domains 27

2.2.2 νPCF domains 30

2.2.3 Least fixed-points 31

2.3 Denotational semantics of νPCF 35

2.3.1 Denoting νPCF types 35

2.3.2 Denoting νPCF terms 36

2.3.3 Relating denotation to operation 37

2.4 Term rewriting and termination 39

2.4.1 Proving rewrite systems terminate 39

2.4.2 Unfold-fold style rewrite rules 42

3 Proof by fixed-point promotion 45

3.1 Overview of fixed-point promotion 45

3.2 Fixed-point promotion vs. cyclic proof 46

3.2.1 Example proof by fixed-point promotion 47

3.2.2 Example cyclic proof 47

4 Preliminaries of fixed-point promotion 51

4.1 Environment variables $\Gamma, \Phi, \mathcal{H}$ 52

4.1.1 Soundness 53

4.2 Many-step rewrite $\xrightarrow{\mathfrak{R}}_+$ 53

4.3 Turning $\xrightarrow{\mathfrak{R}}_+$ into a deterministic algorithm 53

4.4 Preliminary rewrite rules 54

4.4.1 Reduction rewrites 54

4.4.2 Floating pattern matches 55

4.4.3 Removing pattern matches 56

4.4.4 Rewriting to $\perp$ 57

4.4.5 Rewriting sub-terms 58

4.4.6 Using Φ to reduce pattern matches 59

4.4.7 Unfolding fixed-points 60

5 Fusion 63

5.1 How fusion produces fixed-point promoted form 64

5.2 Fusion rules 67

5.2.1 Folding 69

5.3 Constant argument fusion 69

5.4 Fusing a fixed-point into a fixed-point 71

5.5 Constructor fusion 75

5.6 Repeated variable fusion 76

5.7 Free variable fusion 78

5.8 Accumulation fusion 79

5.9 Fact fusion 80

6 Fission 87

6.1 Identity fission 88

6.2 Subterm fission 89

6.3 Constructor fission 91

6.3.1 Auxiliary rewrites for constructor fission 93

6.4 Accumulation fission 95

6.4.1 An auxiliary rewrite for accumulation fission 96

7 A rewriting based theorem prover 99
7.1 νPCF$^{\sqsubseteq}$ . . . 99
7.2 Rewriting νPCF$^{\sqsubseteq}$ to prove theorems . . . 101
7.3 Example proofs . . . 104

8 Discovering fold functions 107
8.1 Motivating examples . . . 107
8.2 Partial solutions . . . 111
8.3 Fold discovery . . . 113
8.4 Examples of fold discovery . . . 116

9 Termination and soundness 123
9.1 Termination . . . 123
9.2 Soundness . . . 125
9.2.1 Traversing into pattern match branches is sound . . . 127
9.2.2 Properties of truncated fixed-points . . . 129
9.2.3 Soundness of fusion . . . 131
9.2.4 Soundness of fission . . . 133

10 Evaluating Elea 135
10.1 Test properties . . . 135
10.2 Compared tools . . . 136
10.2.1 HOSC . . . 136
10.2.2 HipSpec . . . 136
10.2.3 Zeno . . . 137
10.2.4 Oyster/Clam . . . 138
10.3 Table of results . . . 140
10.4 Summary and analysis . . . 149
10.5 Using seq to extend Elea . . . 150

11 Related work 153
11.1 Supercompilation . . . 153
11.2 Deforestation . . . 154
11.3 Generalised partial computation . . . 154
11.4 Improvement theory . . . 155
11.5 Bisimulation . . . 156
11.6 The constructive ω rule . . . 156

12 Conclusion 159

A Term definitions 161

A.1 Functions on `Bool` 161

A.2 Functions on `Nat` 161

A.3 Functions on $\texttt{List}_\tau$ 162

Chapter 1

Introduction

When we manually optimise a computer program, we replace slow, but perhaps more obviously correct code, with faster, but often more complex code. Unfortunately, optimisation through complexity may also introduce software bugs. However, if we are able to prove that our newer, faster code is observationally equivalent to our older, slower code, then we know that our optimisation has not changed the behaviour of our software.

Observational equivalence can show more than just the correctness of optimisations. Any property which can be expressed within our functional language as a boolean term can be proven merely by proving that term equivalent to `True`. Say we have a list sorting function: `sort`, a boolean function which checks if a list is sorted: `sorted`, and a list variable: xs, then we can prove our sorting function always returns a sorted list by showing $\texttt{sorted}\,(\texttt{sort}\,xs) \cong \texttt{True}$, where $\cong$ is our observational equivalence relation.

As we can see, any tool that can prove observational equivalence would be very useful. It would allow us to improve the run-time of a program, knowing that it will not have changed in behaviour. It would also give us a mechanism to prove arbitrary properties of existing code, so that we know our program always produces the correct output, or that it is free of bugs.

Formally, two terms are observationally equivalent if we can replace one with the other in any computer program without changing the program's behaviour. Most languages allow code which can exhibit behaviours other than just returning a value, for instance, modifying a global variable or writing data to a file. These side-effects make reasoning about observational equivalence more difficult; because of this we focus on reasoning about programs in a pure functional language, one in which terms have no behaviours other than returning a value.

Unfortunately, if we are aiming to prove properties for a functional language with non-strict data-types, such as Haskell [35], a lot of these equivalences no longer hold. One of the features of non-strict data-types is that they admit infinite objects. Many properties will fail to terminate given an infinite object as an input, and hence will not be equivalent to `True`. For example, $\texttt{sorted}\,(\texttt{sort}\,xs)$ will not terminate if xs is an infinitely long list, and so $\texttt{sorted}\,(\texttt{sort}\,xs) \cong \texttt{True}$ does not hold.

As well as permitting infinite data structures, non-strict data-types allow non-termination to hide within a data structure, breaking even more equivalences. If `Nat` is a natural number data-type with constructors `0` and `Suc`, and $\perp$ is a non-terminating program,

$\mathtt{Nat}$ being non-strict means that $\bot \ncong \mathtt{Suc}\ \bot$. This invalidates many seemingly obvious properties, like $\mathtt{add}\ x\ (\mathtt{Suc}\ y) \cong \mathtt{Suc}\ (\mathtt{add}\ x\ y)$. Defining $\mathtt{add}$ by $\mathtt{add}\ \mathtt{0}\ y = y$ and $\mathtt{add}\ (\mathtt{Suc}\ x)\ y = \mathtt{Suc}\ (\mathtt{add}\ x\ y)$, this property does not hold for $x = \bot$, as $\mathtt{add}\ \bot\ (\mathtt{Suc}\ y) \cong \bot \ncong \mathtt{Suc}\ \bot \cong \mathtt{Suc}\ (\mathtt{add}\ \bot\ y)$.

In order to recover properties like these, we can scrap equivalence and instead focus on observational approximation. Given terms A and B, A observationally approximates B, written $A \precsim B$, if A can be replaced with B in any terminating program without affecting the program's behaviour. For example, $\mathtt{sorted}\ (\mathtt{sort}\ xs) \precsim \mathtt{True}$ holds, since a terminating program containing $\mathtt{sorted}\ (\mathtt{sort}\ xs)$ must never supply an infinite list for xs. We also have $\mathtt{add}\ x\ (\mathtt{Suc}\ y) \precsim \mathtt{Suc}\ (\mathtt{add}\ x\ y)$, even if $x = \bot$, since $\bot \precsim \mathtt{Suc}\ \bot$. Furthermore, if we do wish to prove an equivalence, we can simply prove approximation in both directions, which is equivalent to equivalence.

When I say "terminates", I really mean "terminates returning a value", since a crashing program terminates but does not return anything. Therefore, this thesis will refer to both crashing and non-terminating terms as "undefined", since neither have a defined result. A good example of a terminating but undefined term is $1/0$, where / is a division operator. In this case we would have $1/0 \precsim A$ for any term A, and hence could replace $1/0$ with A in any defined program without changing its meaning. In general we can think of $\precsim$ as a "less-defined-or-equivalent" ordering between terms.

Proving observational approximation directly is quite difficult, since it is defined in terms of universal quantification over enclosing programs. We can instead prove a stronger property called denotational approximation, $A \sqsubseteq B$, meaning the denotational semantics of term A approximates the denotational semantics of B. Denotational semantics is explained in the next chapter, but for now I will simply state that to prove $A \precsim B$ it is sufficient to prove $A \sqsubseteq B$.

Any language construct whose semantics can be modeled as a non-strict data-type will require approximation to express many of its properties, even if the language it exists within has strict data-types by default. Examples of non-strict structures in otherwise strict languages are sequences in Python [71], or streams in Java [2] version 8. To give an example of such a property, assume `reverse` is a Java stream to stream function which reverses the stream it is given as input. The property $\mathtt{reverse}(\mathtt{reverse}(xs)) \precsim xs$, where xs is a Java stream, does not hold as an equivalence, since an infinite stream will cause non-termination on the left-hand side, and so this property requires approximation to be expressed.

Now that I have explained the motivation for proving properties of approximation rather than equivalence, when dealing with non-strict data-types such as those used by the Haskell language, or any language construct with non-strict semantics, I will discuss how such proofs could be constructed. Equivalence is normally proven using structural induction [57], and there is a long history of automated induction provers for term equivalence in functional languages [38, 14, 18, 21, 33, 45, 64]. To perform such a proof we choose one or more variables in a property, and show that the property holds for any value these variables could take.

Structural induction will work as well for approximation properties as it does for equivalences, but it is not without shortcomings. A weakness specific to structural induction is that it is unable to prove properties which do not have variables amenable to induction.

To give an example of such a property, we can define $\texttt{repeat}\ x = x\ \texttt{::}\ \texttt{repeat}\ x$ and $\texttt{map}\ f\ (x\ \texttt{::}\ xs) = f\ x\ \texttt{::}\ \texttt{map}\ f\ xs$, where $x\ \texttt{::}\ xs$ denotes the element x cons'd onto the list xs. Given these definitions, the following approximation cannot be proven by induction, as it suffers from a lack of variables.

$$\texttt{repeat}\ (\texttt{Suc}\ \texttt{0})\ \sqsubseteq\ \texttt{map}\ \texttt{Suc}\ (\texttt{repeat}\ \texttt{0})$$

Terms such as $\texttt{repeat 0}$ are referred to as codata, and, to prove properties like the above, we can use coinduction [31]. A shortfall of coinduction is that it requires the functions on either side of $\sqsubseteq$ to be productive, also referred to as guarded, which means that their recursive calls can only occur inside constructors. Due to this, we cannot prove $\texttt{sorted}\ (\texttt{sort}\ xs) \sqsubseteq \texttt{True}$ using coinduction, as $\texttt{sorted}$ cannot be productive - it has to recurse over the entire list before producing $\texttt{True}$ if the list is sorted.

Provers which use both induction and coinduction have been developed [46, 49, 48], but only for languages in which no term can be undefined (total languages), and with a strict separation between data-types which admit induction and codata-types which admit coinduction. In comparison, the non-strict data-types in a language such as Haskell can be used to type both data and codata.

Another alternative proof method is fixed-point induction, which is able to prove approximations with a recursive function call top-most on the left-hand side of $\sqsubseteq$. Fixed-point induction can prove both $\texttt{sorted}\ (\texttt{sort}\ xs) \sqsubseteq \texttt{True}$ and $\texttt{repeat}\ (\texttt{Suc 0})\ \sqsubseteq\ \texttt{map}\ \texttt{Suc}\ (\texttt{repeat 0})$. If we were to prove the latter property by fixed-point induction, it would require us to prove:

$$\begin{aligned}\forall f\ .\ \ & f\ (\texttt{Suc 0})\ \sqsubseteq\ \texttt{map}\ \texttt{Suc}\ (\texttt{repeat 0}) \\ & \quad \Rightarrow\quad \texttt{Suc 0}\ \texttt{::}\ f\ (\texttt{Suc 0})\ \sqsubseteq\ \texttt{map}\ \texttt{Suc}\ (\texttt{repeat 0})\end{aligned}$$

This method works by assuming the property holds for any function on the left-hand side, then checks if this property is preserved by one application of the function definition. This is induction over the number of times the function has been unrolled. However, like induction and coinduction, there are properties this method cannot prove. Define $\texttt{double 0} = \texttt{0}$ and $\texttt{double}\ (\texttt{Suc}\ x)\ y = \texttt{Suc}\ (\texttt{Suc}\ (\texttt{double}\ x))$ and try to prove $\texttt{add}\ x\ x \sqsubseteq \texttt{double}\ x$ by fixed-point induction. You will get stuck attempting to show the property below, as it does not hold.

$$\forall f\ .\ \ (\forall x\ .\ f\ x\ x\ \sqsubseteq\ \texttt{double}\ x)\ \Rightarrow\ (\forall x\ .\ f\ x\ (\texttt{Suc}\ x)\ \sqsubseteq\ \texttt{Suc}\ (\texttt{double}\ x))$$

To invalidate this property we need only choose f to be a function which acts like $\texttt{add}$ only when given identical arguments and returns $\texttt{0}$ otherwise.

These methods: induction, coinduction and fixed-point induction, are all types of cyclic [8] proof[1], as they assume the goal in order to prove the goal, but with some restriction on how this assumption can be used. Induction requires some decreasing measure on variables, coinduction requires productivity, and fixed-point induction requires the generalisation of a recursive function to a function variable. As we have seen, all three have their own

[1]They are also referred to as circular [59] proof methods, but in this thesis I use the term cyclic, as to me, circular evokes the logical fallacy of circular reasoning.

unique drawbacks, but there is one shared drawback to all forms of cyclic proof, the problem of generalisation [30].

Generalising a property is the act of strengthening it in order to make it easier to prove. This may seem counter-intuitive, as any proof of a stronger property will also prove the weaker, but not vice versa, hence the stronger property may have fewer proofs, and hence it should be harder to find a proof for a stronger property. In cyclic proofs however, since we assume the property in order to prove it, a stronger property leads to a stronger assumption, and stronger assumptions mean easier proofs. The generalisation problem refers to the difficulty of choosing how to strengthen a property such that we can prove it cyclically, without strengthening it so much that it no longer holds.

As an example of the generalisation problem, take the following property

$$\texttt{add}\, x\, (\texttt{Suc}\, x) \sqsubseteq \texttt{Suc}\, (\texttt{add}\, x\, x)$$

Let's try to prove this cyclically. We first assume it holds for all x, then try to prove it holds for all x. Starting with case analysis on x, the case when $x = \texttt{0}$ is trivial, but not so for the case when $x = \texttt{Suc}\, x'$, for some x', which requires us to show

$$\texttt{Suc}\, (\texttt{add}\, x'\, (\texttt{Suc}\, (\texttt{Suc}\, x'))) \sqsubseteq \texttt{Suc}\, (\texttt{Suc}\, (\texttt{add}\, x'\, (\texttt{Suc}\, x')))$$

Our cyclic assumption is not applicable to the above, so we are stuck. Now let's try this proof again. First we generalise the second x on each side of $\sqsubseteq$ to some new y, yielding

$$\texttt{add}\, x\, (\texttt{Suc}\, y) \sqsubseteq \texttt{Suc}\, (\texttt{add}\, x\, y)$$

Again we do case analysis on x, but the case when $x = \texttt{Suc}\, x'$ is now

$$\texttt{Suc}\, (\texttt{add}\, x'\, (\texttt{Suc}\, y)) \sqsubseteq \texttt{Suc}\, (\texttt{Suc}\, (\texttt{add}\, x'\, y))$$

Our cyclic assumption is now applicable as a rewrite to either side of this property[2]; so let's use it to rewrite $\texttt{add}\, x'\, (\texttt{Suc}\, y)$ to $\texttt{Suc}\, (\texttt{add}\, x'\, y)$ on the left-hand side, which gives us

$$\texttt{Suc}\, (\texttt{Suc}\, (\texttt{add}\, x'\, y)) \sqsubseteq \texttt{Suc}\, (\texttt{Suc}\, (\texttt{add}\, x'\, y))$$

This property trivially holds by the reflexivity of $\sqsubseteq$, so we have proven our goal. This cyclic proof could be either induction, since in applying our assumption we unified x with the structurally smaller value x', coinduction, since we applied the assumption inside a constructor and hence fulfil the productivity requirement, or fixed-point induction, since we could have generalised `add` on the left-hand side of $\sqsubseteq$ to some new f and the proof would still be correct.

As you can see, the generalisation of those xs made our cyclic assumption strong enough that it was applicable in our proof, but the problem is: how do we build a theorem

[2]In general, if we have $A \sqsubseteq B$ and are trying to prove $C \sqsubseteq D$, we can rewrite any instance of A to B in C, or any instance of B to A within D, to yield a property sufficient to prove $C \sqsubseteq D$. When proving equivalence properties we can use our cyclic assumption as a rewrite in either direction on either side of the equivalence and yield a necessary and sufficient property. Proving approximation enforces a directionality on our rewriting, and only yields a sufficient property.

prover which can recognise this as the required generalisation? The difficulty of this question prompted me to explore a method which does not suffer from this issue: term rewriting; in particular the unfold-fold style [16], which includes methods such as partial evaluation [34, 23], supercompilation [68, 66, 6], deforestation [72, 51], and fixed-point fusion [54]. When I say that we can prove a property by term rewriting, as opposed to cyclically, I mean that we rewrite the terms within the property to the point that a cyclic method is no longer required to complete the proof.

The statement that term rewriting techniques do not suffer from the generalisation problem is something which might sound odd to those familiar with the unfold-fold literature, since it is littered with the word generalisation [65, 42, 6]. However, the problem of generalisation for rewriting systems is far simpler than the equivalent problem for cyclic provers, as the generalisation does not have to be matched in the enclosing proof context.

To explain what I mean by this, take the aforementioned property $\texttt{add}\ x\ (\texttt{Suc}\ x) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ x)$. As with cyclic proof, when given the term $\texttt{add}\ x\ (\texttt{Suc}\ x)$, rewriting systems must generalise the second x to some new y, but detecting such a generalisation is a solved problem [65]. However, if we are approaching this as a cyclic prover we must also generalise the right-hand side of the property in such a way that it matches the generalisation of $\texttt{add}\ x\ (\texttt{Suc}\ x)$. This is to say we must somehow detect that the second x in $\texttt{Suc}\ (\texttt{add}\ x\ x)$ is the one that should be replaced by the same y, and while there are heuristics which can solve this particular case [30], they would not work for a property like $\texttt{add}\ x\ (\texttt{double}\ x) \sqsubseteq \texttt{add}\ (\texttt{double}\ x)\ x$, which for non-strict data-types does not look to admit any generalisation at all! Simply put, generalising a single term such that unfold-fold rewriting is possible is easier and more applicable than generalising an entire property such that we can use a cyclic proof method.

As we have seen, rewriting based proof does not suffer from the generalisation problem to the same extent as cyclic proof methods, since they consider terms separately from the property they exist within. However, this advantage of rewriting can also be a disadvantage when keeping the enclosing property gives you information needed to complete your proof. I refer to this as the lemma discovery problem and, to give an example of what I mean, let's consider the property $\texttt{add}\ x\ x \sqsubseteq \texttt{double}\ x$. Automated cyclic provers I have seen, if given this property, in the case when $x = \texttt{Suc}\ x'$, will reach the sub-goal

$$\texttt{add}\ x'\ (\texttt{Suc}\ x') \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x'\ x')$$

If possible, they will detect the generalisation just discussed, yielding the easily proven property

$$\texttt{add}\ x'\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x'\ y)$$

A supercompiler, to choose a well-defined variant of unfold-fold rewriting, could show $\texttt{add}\ x\ x \sqsubseteq \texttt{double}\ x$ by rewriting $\texttt{add}\ x\ x$ to a term α-equal to $\texttt{double}\ x$, and then appealing to the reflexivity of $\sqsubseteq$. In order to do this, however, this supercompiler would have to discover the rewrite

$$\texttt{add}\ x'\ (\texttt{Suc}\ y) \longrightarrow \texttt{Suc}\ (\texttt{add}\ x'\ y)$$

This matches the $\texttt{add}\ x'\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x'\ y)$ property our cyclic prover would have to discover, and, in the cases given for both cyclic proof and supercompilation, we can think of this property as the lemma we must discover in order to complete the proof. This is easy for cyclic proof, as the lemma is a generalisation of a sub-goal, but not so for supercompilation, as it does not consider the enclosing property, and so would have to infer this lemma/rewrite another way. In summary, separating terms from the properties they exist within makes generalisation an easier problem for term rewriting, but lemma discovery harder, with the opposite being true for cyclic proof.

This thesis describes the design and implementation of an automated theorem prover for properties of terms in a pure, call-by-name, functional language with non-strict data-types. This tool, which I have called Elea, proves denotational approximation between terms, so that its domain of properties is much larger than if it were only able to prove equivalence, and which enables it to prove predicates like $\texttt{sorted}\,(\texttt{sort}\,xs) \sqsubseteq \texttt{True}$.

Elea proves these properties using a term rewriting system similar to supercompilation, chosen so that it does not suffer from the generalisation problem to the extent that a cyclic prover would. It also does not have the individual weaknesses of each variant of aforementioned cyclic proof; unlike induction it can prove properties of codata, and unlike coinduction it can prove properties of non-productive functions. To develop this rewriting system, I have invented techniques which perform rewrites equivalent to the lemma discovery steps a cyclic prover could make, but without requiring the enclosing property, negating the advantage cyclic provers have over term rewriting provers.

I have compared my tool to two automated equivalence provers based on induction, HipSpec [18] and Zeno [64], and found many properties Elea could prove which these systems could not, only one property Zeno could prove over Elea, but quite a few properties HipSpec could prove over Elea. However, both Zeno and HipSpec prove equivalences for only total terms, ignoring non-termination; Elea proves these properties as approximations, or equivalences where possible, and does not require that terms be total. I have also compared my tool to the supercompilation based equivalence prover HOSC [40], and found Elea able to prove strictly more properties.

1.1 Contributions

The major contribution of this thesis is the design and implementation of Elea, an automated theorem prover for properties of denotational approximation between terms in a pure, call-by-name, functional language with non-strict data-types. This contribution can be broken into six major points. Within this breakdown I will use the term relation $A \xrightarrow{\sqsubseteq}_+ B$ to means that Elea can rewrite A to B, such that $A \sqsubseteq B$, and all terms in this section are defined at the end of this chapter in Definition 1.1

1. I developed the truncation fusion style of unfold-fold rewrite rules (Chapter 5), which simplify the problem of generalisation, allowing for rewrites such as $\texttt{half}\ (\texttt{add}\ x\ x) \xrightarrow{\sqsubseteq}_+ \texttt{True}$. This rewrite cannot be achieved through traditional supercompilation.

2. I adapted truncation fusion, and unfold-fold rewriting in general, to prove the unreachability of branches within recursive functions, a technique I call fact fusion (Section 5.9). An example of this is the rewrite $\texttt{or}\ (\texttt{eq}\ x\ y)\ (\texttt{eq}\ y\ x) \xrightarrow{\sqsubseteq}_+$

or (eq x y) True, in which every branch returning False within eq y x was shown to be unreachable, so the entire term eq y x could be rewritten to True.

3. I invented a set of rewrite rules I collectively refer to as fission (Chapter 6), which allow Elea to extract contexts from recursive functions. These fission rewrites are essential for many larger proofs. For example, the fission rewrite add x (Suc y) $\xrightarrow{\sqsubseteq}_+$ Suc (add x y) is used to show half (add x x) $\sqsubseteq$ True.

4. I created the fold-discovery technique for aiding in the completion of truncation fusion steps (Chapter 8). It is this method which allows Elea to automatically perform rewrites such as sorted (isort xs) $\xrightarrow{\sqsubseteq}_+$ True and it-rev (it-rev xs [ ]) [ ] $\xrightarrow{\sqsubseteq}_+$ xs.

5. I identified a non-cyclic proof method which naturally complements unfold-fold style rewriting - the least fixed-point principle (Chapter 3). Using this technique, Elea is able to prove properties which cannot be shown by rewriting alone, such as eq x y $\sqsubseteq$ eq y x.

6. I discovered a new method for proving soundness of an unfold-fold style rewriting system (Section 9.2). This proof method, called truncation fusion, is based on an existing technique called truncation induction. This was necessary as existing methods did not allow me to prove all of Elea's rewrite techniques sound, an issue which is discussed in Section 11.4.

1.2 Thesis outline

The remainder of this thesis is structured as follows:

- Chapter 2 gives the relevant background material for this work, including a formal description of the input language to Elea, denotational semantics, and an introduction to term rewriting systems.
- Chapter 3 gives a high-level overview of how Elea proves approximation properties using a rewriting system, a method I call fixed-point promotion.
- Chapter 4 describes the preliminaries of Elea's rewrite system, including the notation I use and the simplest of its rewrite rules, such as beta reduction.
- Chapter 5 gives my unfold-fold style rewrite rules, which I collectively call fusion rewrites.
- Chapter 6 explains what I call fission rewrite rules, which allow Elea to extract contexts from recursive functions.
- Chapter 7 formally describes the theorem prover I have developed around my term rewriting system.
- Chapter 8 details the fold discovery rewrite technique, which affords Elea its most complex rewrites.

- Chapter 9 proves Elea both sound and terminating. The proof of termination is by an established method called the homeomorphic embedding [47]. The proof of soundness is by denotational semantics, including a novel method based on a technique called truncation induction [50].
- Chapter 10 presents a comparison between Elea and the tools Zeno, HipSpec, HOSC, and Oyster/Clam.
- Chapter 11 describes the existing work related to Elea
- Chapter 12 concludes this thesis.

Definition 1.1 (Term definitions).

$$
\begin{array}{lcll}
\texttt{app}\;[\,]\;ys & = & ys & \text{(list append)} \\
\texttt{app}\;(x :: xs)\;ys & = & x :: \texttt{app}\;xs\;ys & \\[4pt]
\texttt{rev}\;[\,] & = & [\,] & \text{(list reverse)} \\
\texttt{rev}\;(x :: xs) & = & \texttt{app}\;(\texttt{rev}\;xs)\;[x] & \\[4pt]
\texttt{it-rev}\;[\,]\;ys & = & ys & \text{(iterative list reverse)} \\
\texttt{it-rev}\;(x :: xs)\;ys & = & \texttt{it-rev}\;xs\;(x :: ys) & \\[4pt]
\texttt{half}\;\texttt{0} & = & \texttt{0} & \text{(halve a number)} \\
\texttt{half}\;(\texttt{Suc}\;x) & = & \texttt{0} & \\
\texttt{half}\;(\texttt{Suc}\;(\texttt{Suc}\;x)) & = & \texttt{Suc}\;(\texttt{half}\;x) & \\[4pt]
\texttt{not}\;\texttt{False} & = & \texttt{True} & \text{(logical not)} \\
\texttt{not}\;\texttt{True} & = & \texttt{False} & \\[4pt]
\texttt{and}\;\texttt{False}\;b & = & \texttt{False} & \text{(logical and)} \\
\texttt{and}\;\texttt{True}\;b & = & b &
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{lq}\;\texttt{0}\;\texttt{0} & = & \texttt{True} & \text{(less-than-or-equal)} \\
\texttt{lq}\;\texttt{0}\;(\texttt{Suc}\;y) & = & \texttt{True} & \\
\texttt{lq}\;(\texttt{Suc}\;x)\;\texttt{0} & = & \texttt{False} & \\
\texttt{lq}\;(\texttt{Suc}\;x)\;(\texttt{Suc}\;y) & = & \texttt{lq}\;x\;y & \\[4pt]
\texttt{sorted}\;[\,] & = & \texttt{True} & \text{(sortedness check)} \\
\texttt{sorted}\;[x] & = & \texttt{True} & \\
\texttt{sorted}\;(x :: y :: ys) & = & \texttt{and}\;(\texttt{le}\;x\;y) & \\
 & & \quad(\texttt{sorted}\;(y :: ys)) & \\[4pt]
\texttt{insert}\;n\;[\,] & = & [n] & \\
\texttt{insert}\;n\;(x :: xs) & = & n :: x :: xs \qquad \text{if}\;\texttt{le}\;n\;x & \\
\texttt{insert}\;n\;(x :: xs) & = & x :: \texttt{insert}\;n\;xs & \\[4pt]
\texttt{isort}\;[\,] & = & [\,] & \text{(insertion sort)} \\
\texttt{isort}\;(x :: xs) & = & \texttt{insert}\;x\;(\texttt{isort}\;xs) &
\end{array}
$$

Chapter 2

Background

The previous chapter explained the aim of this thesis: building a proof system for observational approximation between terms in a functional language. This chapter describes the existing literature drawn upon to develop this system.

(Section 2.1)	Describes the programming language my Elea tool proves approximation properties for - νPCF. It is the call-by-name PCF [58] language with user defined algebraic data-types, pattern matching and non-strict constructors. This section gives its syntax, typing rules, and operational semantics.
(Section 2.2)	Introduces the theory underlying the theory used to prove my term rewriting system sound - domain theory. This section describes what a domain is, and the all important notions of continuous functions and least fixed-points.
(Section 2.3)	Uses the theory in the previous section to describe the theory used to show Elea is sound - denotational semantics. The denotational semantics of a programming language is a mapping from the constructs of that language to mathematical objects which we can then reason about. I use this to show that the rewriting system described in this thesis preserves denotational approximation. As outlined at the end of this section, denotational approximation implies observational approximation, which proves that my rewrite system preserves observational approximation, as was its aim.
(Section 2.4)	Explains term rewriting systems, and the technique used to ensure Elea terminates, which is called the homeomorphic embedding. This section goes on to explain the general class of unfold-fold rewriting techniques, which Elea's fusion steps are an instance of.

2.1 νPCF

νPCF is the programming language my tool proves properties about. It is Plotkin's call-by-name language of Programmable Computable Functions (PCF) [58] extended to have user defined data-types with non-strict constructors, along with a construct I call a truncated fixed-point - a recursive term that can only be unrolled a finite number of times. This section breaks down the description of this language into the following sections.

(Section 2.1.1)	Syntax
(Section 2.1.2)	Typing rules
(Section 2.1.3)	Evaluation rules (operational semantics)
(Section 2.1.4)	Syntactic sugar for νPCF, useful constructs defined in terms of existing syntax, and hence requiring no extra typing or evaluation rules.
(Section 2.1.5)	The $\text{fold}_{\boldsymbol{T}}\langle ... \rangle$ syntax, which represents the "fold" function [29, 52] of a given data-type. This is a construct which generalises a very common form of recursive function, and is the foundation of one of Elea's rewriting techniques, fold discovery from Chapter 8. It is also syntactic sugar but requires enough description that it got its own section.

2.1.1 νPCF syntax

Here I define the term and type grammar of νPCF, along with the recursive equations defining the common data-types used throughout this thesis, and the grammar of νPCF term contexts. I also give the definition of freeVars, an operator which returns the free variables of a given νPCF term. In this thesis I use the word operator to mean a function in my meta-language, which is to say a function implemented within the Elea tool itself. The word function will refer only to functions implemented in the νPCF input language of Elea.

I have included a non-standard construct in the grammar of νPCF, truncated fixed-points. These are fixed-points which can only be unfolded a finite number of times, where this number is given by the meta-variable which annotates them. The input language to Elea does not include truncated fixed-points, they are a construct introduced and subsequently removed by my truncated fusion rewrite rule (Chapter 5), and are used to ensure the soundness of this rewrite.

Definition 2.1 (νPCF types).

$$\tau \ ::= \ \boldsymbol{T} \mid \tau \to \tau$$

νPCF types, ranged over by τ, are either data-types or function types. Data-types in νPCF, ranged over by $\boldsymbol{T}$, are defined as a set of potentially recursive equations of the form

$$\boldsymbol{T} = c_1 \mid ... \mid c_n \qquad \text{where} \qquad c \ ::= \ (\boldsymbol{T}_1, ..., \boldsymbol{T}_m)$$

Here c gives a constructor definition - a list of arguments. This list can be empty, written (). Unlike a real-world functional language, such as Haskell or Standard ML, the arguments to constructors can only be of data type ($\boldsymbol{T}$), rather than any type (τ). This restricts νPCF to what is referred to as polynomial data-types, but since all of the properties I designed Elea to prove only involve such data-types, the extra background theory required to support non-polynomial data-types would be an unnecessary complication.

Definition 2.2 (Common data-types).

$$\begin{array}{lcl}
\texttt{Unit} & = & () \\
\texttt{Bool} & = & () \mid () \\
\texttt{Nat} & = & () \mid \texttt{Nat} \\
\texttt{List}_\tau & = & () \mid (\tau, \texttt{List}_\tau) \\
\texttt{Tree}_\tau & \overset{\text{def}}{=} & () \mid (\texttt{Tree}_\tau, \tau, \texttt{Tree}_\tau)
\end{array}$$

Here are the equations defining the data-types used throughout this thesis. The type argument, τ, in the definition of $\texttt{List}_\tau$ and $\texttt{Tree}_\tau$ is not polymorphism, but just a macro for generating definitions for lists of different types. There is implicitly a new recursive equation for every different argument provided.

Definition 2.3 (νPCF syntax).

$$\begin{array}{lcll}
A, ..., Z & ::= & x & \text{variable} \\
& \mid & F\ A & \text{function application} \\
& \mid & \text{con}_{\mathbb{N}}\langle \boldsymbol{T} \rangle & \text{constructor} \\
& \mid & \text{fix}\,(F) & \text{least fixed-point} \\
& \mid & \text{fix}^a\,(F) & \text{truncated fixed-point} \\
& \mid & \text{fn}\ x : \tau.\ F & \text{function abstraction} \\
& \mid & \text{else} & \text{default pattern} \\
& \mid & \text{case}\ M\ \text{of}\ P_1 \to B_1 \ldots P_n \to B_n & \text{pattern match}
\end{array}$$

Lower-case italic letters range over νPCF variables, excluding a, b, and c, which are meta-variables of type $\mathbb{N}$. These meta-variables are just extra symbols within this language grammar, used to tag truncated fixed-points such that fix^a is distinguishable from fix^b. Upper-case italic letters range over terms, and to aid readability I will commonly use F, G, and H for function terms, and A, B, and C for non-function terms. I will use K for constructors, terms of the shape $\text{con}_i\langle \boldsymbol{T} \rangle$ representing the ith constructor of the data-type $\boldsymbol{T}$, which is to say the ith injector into the lifted disjoint union represented by $\boldsymbol{T}$. In pattern matches I use P for the patterns, M for the matched term, and often B for the branches. Pattern terms P_i in the definition of pattern matches will always have the shape else or $K\ x_1...x_n$, as enforced by the typing rules for νPCF. These rules also enforce that else can only appear as a pattern term. I will very often elide the type of a function abstraction, as it can usually be easily deduced from the context.

Definition 2.4 (νPCF contexts).

$$\begin{array}{ll}
\mathcal{C} ::= \square \mid x \mid \text{con}_{\mathbb{N}}\langle \boldsymbol{T} \rangle \mid \text{fix}\,(\mathcal{C}) \mid \text{fix}^a\,(\mathcal{C}) \mid \text{fn}\ x : \tau.\ \mathcal{C} \mid \mathcal{C}\ \mathcal{C}' \mid \text{else} \\
\quad \mid \text{case}\ \mathcal{C}\ \text{of}\ P_1 \to \mathcal{C}_1 \ldots P_n \to \mathcal{C}_n
\end{array}$$

A term context is a term with a piece missing. This ”hole” is denoted “$\Box$”, and applying the context $\mathcal{C}$ to the term A, written $\mathcal{C}[A]$, denotes the replacement of all occurrences of $\Box$ within $\mathcal{C}$ with A.

Definition 2.5 ($\mathsf{freeVars}\,(...)$).

$$
\begin{aligned}
&\mathsf{freeVars}\,(x) && \stackrel{\text{def}}{=} \{\ x\ \}\\
&\mathsf{freeVars}\,(K) && \stackrel{\text{def}}{=} \emptyset\\
&\mathsf{freeVars}\,(\text{else}) && \stackrel{\text{def}}{=} \emptyset\\
&\mathsf{freeVars}\,(F\ A) && \stackrel{\text{def}}{=} \mathsf{freeVars}\,(F) \cup \mathsf{freeVars}\,(A)\\
&\mathsf{freeVars}\,(\text{fn}\ x.\ F) && \stackrel{\text{def}}{=} \mathsf{freeVars}\,(F) - \{x\}\\
&\mathsf{freeVars}\,(\text{fix}\,(F)) && \stackrel{\text{def}}{=} \mathsf{freeVars}\,(F)\\
&\mathsf{freeVars}\,(\text{fix}^a\,(F)) && \stackrel{\text{def}}{=} \mathsf{freeVars}\,(F)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{freeVars}\big(\text{case}\ M\ \text{of}\ p_1 \to B_1\ ...\ p_n \to B_n\big) \stackrel{\text{def}}{=}\\
&\quad \mathsf{freeVars}\,(M) \cup \textstyle\bigcup_i \left(\mathsf{freeVars}\,(B_i) - \mathsf{freeVars}\,(p_i)\right)
\end{aligned}
$$

The operator $\mathsf{freeVars}$ returns the free variables of a given term or pattern.

2.1.2 Typing νPCF terms

The rules in Definition 2.9 inductively define a well-typed νPCF term, referring to a type environment Γ, described by Definition 2.7. These rules make use of the $\mathsf{conArgs}$ operator (Definition 2.6) and the $\lhd$ operator (Definition 2.8).

Definition 2.6 ($\mathsf{conArgs}$).

$$
\begin{aligned}
&\mathsf{conArgs}_i\,\boldsymbol{T} \stackrel{\text{def}}{=} (\tau_1, ..., \tau_n)\\
&\qquad \text{where}\quad (\boldsymbol{T} = c_1 \mid ... \mid c_m) \quad \text{is a data-type definition}\\
&\qquad\qquad\quad\ (\tau_1, ..., \tau_n) = c_i
\end{aligned}
$$

The $\mathsf{conArgs}$ operator takes a constructor index $i \in \mathbb{N}$, and a data-type $\boldsymbol{T}$, and returns the argument types of its ith constructor by looking them up in the data-type definition.

Example 2.1 (Using $\mathsf{conArgs}$).

$$
\begin{aligned}
&\mathsf{conArgs}_1\,\texttt{Nat} && = ()\\
&\mathsf{conArgs}_2\,\texttt{Nat} && = (\texttt{Nat})\\
&\mathsf{conArgs}_2\,\texttt{List}_{\texttt{Nat}} && = (\texttt{Nat}, \texttt{List}_{\texttt{Nat}})
\end{aligned}
$$

The first constructor of `Nat`, `0`, has no arguments, while the second constructor, `Suc`, has a single `Nat` typed argument. The second constructor, `::`, of lists of natural numbers takes two arguments, a `Nat` and another such list.

Definition 2.7 (Type environment Γ). A type environment, Γ, is a partial function from variables to types. $\emptyset$ denotes the empty type environment, which is undefined at every input. $\Gamma[x \mapsto \tau]$ is the type environment that returns τ given the variable x, and behaves like Γ otherwise.

Definition 2.8 (The $\lhd$ operator).

$$\begin{aligned}
\Gamma \lhd \text{ else } &\overset{\text{def}}{=} \Gamma \\
\Gamma \lhd \text{ con}_i\langle \boldsymbol{T} \rangle\, x_1 ... x_n &\overset{\text{def}}{=} \Gamma[x_1 \mapsto \tau_1]...[x_n \mapsto \tau_n] \\
&\quad \text{where } (\tau_1, ..., \tau_n) = \mathsf{conArgs}_i\, \boldsymbol{T}
\end{aligned}$$

The $\lhd$ operator takes a type environment and the pattern from a pattern match, and returns the type environment with the bindings of matched pattern added to it. It is used to type the branches of a pattern match, as the variables bound by its pattern, P, will be scoped locally to that branch.

Definition 2.9 (Typing νPCF terms).

$$(:_{\text{var}}) \quad \frac{}{\Gamma \vdash x : \Gamma\, x} \quad \text{if } x \in \mathsf{dom}\,(\Gamma) \qquad (:_{\text{app}}) \quad \frac{\Gamma \vdash F : \tau \to \tau' \qquad \Gamma \vdash A : \tau}{\Gamma \vdash F\ A : \tau'}$$

$$(:_{\text{fix}}) \quad \frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \text{fix}\,(F) : \tau} \qquad (:_{\text{tfix}}) \quad \frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \text{fix}^a\,(F) : \tau} \qquad (:_{\text{fn}}) \quad \frac{\Gamma[x \mapsto \tau] \vdash A : \tau'}{\Gamma \vdash \text{fn } x : \tau.\ A : \tau \to \tau'}$$

$$(:_{\text{con}}) \quad \frac{}{\Gamma \vdash \text{con}_i\langle \boldsymbol{T} \rangle : \tau_1 \to ... \to \tau_n \to \boldsymbol{T}} \qquad \text{if } (\tau_1, ..., \tau_n) = \mathsf{conArgs}_i\, \boldsymbol{T}$$

$$(:_{\text{case}}) \quad \frac{\begin{array}{ccc} & \Gamma \lhd P_1 \vdash P_1 : \boldsymbol{T} & \Gamma \lhd P_1 \vdash B_1 : \tau \\ & \vdots & \vdots \\ \Gamma \vdash M : \boldsymbol{T} & \Gamma \lhd P_n \vdash P_n : \boldsymbol{T} & \Gamma \lhd P_n \vdash B_n : \tau \end{array}}{\Gamma \vdash \text{case } M \text{ of } P_1 \to B_1 ... P_n \to B_n : \tau}$$

$$(:_{\text{else}}) \quad \frac{}{\Gamma \vdash \text{else} : \boldsymbol{T}} \qquad \text{if occurring as a pattern in a case...of term}$$

These typing rules have the form $\Gamma \vdash A : \tau$ denoting term A to have type τ in environment Γ. The notation $A : \tau$ is shorthand for $\emptyset \vdash A : \tau$. The shape of type environments Γ is detailed in Definition 2.7 and the $\lhd$ operator in Definition 2.8.

2.1.3 Operational semantics of νPCF

The operational semantics of a language formalises the result of executing a closed term (a "program") from the language, using a computer. The judgement $A \Downarrow_\tau V$ denotes that program $A : \tau$ terminates with value V. The judgement $A \Downarrow$ is shorthand for $\exists V \,.\, A \Downarrow_\tau V$, viz. that A is a terminating program.

The $\Downarrow_\tau$ relation is given inductively by the rules in Definition 2.12 and makes use of the $\mathsf{findPattern}$ operator from Definition 2.11. Values are a sub-set of terms which cannot be evaluated further, described in Definition 2.10.

It is worth noting that the operational semantics of truncated fixed-points are not particularly necessary, as Elea does not allow truncated fixed-points within its input language, I include them to preserve the desired properties of νPCF's denotational semantics, given later in Section 2.1.3.

Definition 2.10 (Values in νPCF).

$$V ::= K\ A_1 ... A_n \ \mid\ \text{fn}\ x : \tau .\ A$$

A value is a term which can be evaluated no further. That constructor values have term arguments, rather than value arguments, is what gives νPCF non-strict constructors.

Definition 2.11 (The $\mathsf{findPattern}$ operator).

$$\mathsf{findPattern}_i\ (P_1, ..., P_n)\ \stackrel{\text{def}}{=}\ \min_k \big(P_k = \text{con}_i \langle \boldsymbol{T} \rangle\ ... \lor P_k = \text{else}\big)$$

Executing pattern match terms requires looking up the branch we have matched to. This is what the $\mathsf{findPattern}_i$ operator does - find the first pattern in a list which has the given constructor index i, or is an else pattern.

Definition 2.12 (Operational semantics of νPCF).

$$(\Downarrow_{\text{val}})\ \frac{}{V \Downarrow_\tau V}\ \text{if}\ V : \tau \qquad (\Downarrow_{\text{cbn}})\ \frac{F \Downarrow_{\tau \to \tau'} \text{fn}\ x : \tau .\ B \qquad B[A/x] \Downarrow_{\tau'} V}{F\ A \Downarrow_{\tau'} V}$$

$$(\Downarrow_{\text{fix}})\ \frac{F\ (\text{fix}\,(F)) \Downarrow_\tau V}{\text{fix}\,(F) \Downarrow_\tau V} \qquad (\Downarrow_{\text{tfix}})\ \frac{F\ (\text{fix}^a\,(F)) \Downarrow_\tau V}{\text{fix}^{a+1}\,(F) \Downarrow_\tau V}$$

$$(\Downarrow_{\text{case}})\ \frac{M \Downarrow_{\boldsymbol{T}} \text{con}_i \langle \boldsymbol{T} \rangle\ A_1 ... A_n \qquad B_k[A_1/x_1]...[A_n/x_n] \Downarrow_\tau V}{\text{case}\ M\ \text{of}\ P_1 \to B_1\ ...\ P_m \to B_m \Downarrow_\tau V}$$

$$\text{where}\quad k = \mathsf{findPattern}_i\ (P_1, ..., P_m)$$
$$K\ x_1 ... x_n = P_k$$

2.1.4 Syntactic sugar for νPCF

This section defines some additional syntax for νPCF using existing language constructs. Defining them this way means we do not have to add any typing and evaluation rules.

Most functional languages have some syntax which allows user defined terms, normally denoted let. νPCF has no such construct, and so to aid readability throughout this thesis I will define many term `names` as shorthand for νPCF terms. Some of these names may have subscripted arguments, in which case they act as syntactic macros. These names do not exist within the νPCF language, and are just synonyms for terms. This may seem fairly convoluted, but it greatly simplifies the presentation of the various rewrite rules in this thesis, not least by the fact that they are not required to carry around a list of name definitions in their environment.

Definition 2.13 gives a set of such term macros for the constructors of the data-types given in Definition 2.2 on page 19. Appendix A gives the full list of the macros which define the functions used throughout this thesis.

The rest of this section defines more complex syntatic sugar: the $\perp_\tau$, if...then...else, seq, and assert constructs. It then defines two extensions of existing constructs: multiple term pattern matches and multiple variable function abstractions.

Definition 2.13 (Constructors of Definition 2.2).

$$\begin{aligned}
\texttt{unit} &\overset{\text{def}}{=} \text{con}_1\langle\texttt{Unit}\rangle \\[1em]
\texttt{False} &\overset{\text{def}}{=} \text{con}_1\langle\texttt{Bool}\rangle \\
\texttt{True} &\overset{\text{def}}{=} \text{con}_2\langle\texttt{Bool}\rangle \\[1em]
\texttt{0} &\overset{\text{def}}{=} \text{con}_1\langle\texttt{Nat}\rangle \\
\texttt{Suc} &\overset{\text{def}}{=} \text{con}_2\langle\texttt{Nat}\rangle \\[1em]
\texttt{[]}_\tau &\overset{\text{def}}{=} \text{con}_1\langle\texttt{List}_\tau\rangle \\
\texttt{Cons}_\tau &\overset{\text{def}}{=} \text{con}_2\langle\texttt{List}_\tau\rangle \\
A ::_\tau B &\overset{\text{def}}{=} \texttt{Cons}_\tau\, A\, B \\
\texttt{[}A\texttt{]}_\tau &\overset{\text{def}}{=} A ::_\tau \texttt{[]} \\[1em]
\texttt{Leaf}_\tau &\overset{\text{def}}{=} \text{con}_1\langle\texttt{Tree}_\tau\rangle \\
\texttt{Node}_\tau &\overset{\text{def}}{=} \text{con}_2\langle\texttt{Tree}_\tau\rangle
\end{aligned}$$

Here we name the constructors of the data-types from Definition 2.2, so that we can refer to them by these names when describing terms, rather than using our anonymous definition syntax. In the case of list and tree constructors, I will often omit the type argument when its value is clear from the context.

Definition 2.14 ($\perp_\tau$).

$$\perp_\tau \overset{\text{def}}{=} \text{fix}\,(\text{fn}\ x : \tau.\, x)$$

The syntax $\bot_\tau$ represents the undefined term of type τ. Any non-terminating term would do as its definition, but the one chosen is standard in the literature.

Definition 2.15 (if...then...else).

$$\text{if } M \text{ then } A \text{ else } B \quad \overset{\text{def}}{=} \quad \text{case } M \text{ of } \begin{array}{l} \texttt{True} \to A \\ \texttt{False} \to B \end{array}$$

An if...then...else term is a synonym for a pattern match on a boolean value.

Definition 2.16 (seq...in).

$$\text{seq}\, M \text{ in } A \quad \overset{\text{def}}{=} \quad \text{case } M \text{ of else} \to A$$

The term seq M in A evaluates M then returns A. The difference between this term and just A is that, if M is undefined, then seq M in A is also undefined, regardless of the value of A.

Definition 2.17 (assert).

$$\text{assert}_\tau\ p \leftarrow M \text{ in } A \quad \overset{\text{def}}{=} \quad \text{case } M \text{ of } \begin{array}{l} p \to A \\ \text{else} \to \bot_\tau \end{array}$$

I define an assertion to be a pattern match which is undefined unless a specific match occurs. I use assertions to express pattern matches which have already been made, since if we know $M \equiv p$ we have that $A \equiv$ assert $p \leftarrow M$ in A.

Definition 2.18 (Multiple term pattern matching).

$$\begin{pmatrix} \text{case } M_1, M_2, ... \text{ of} \\ \quad p_1^1, p_1^2, ... \to A_1\ ... \\ \quad p_n^1, p_n^2, ... \to A_n \end{pmatrix} \quad \overset{\text{def}}{=} \quad \begin{pmatrix} \text{case } M_1 \text{ of} \\ \quad p_1^1 \to \begin{pmatrix} \text{case } M_2, ... \text{ of} \\ \quad p_1^2, ... \to A_1\ ... \\ \quad p_n^2, ... \to A_n \end{pmatrix} \\ \quad p_n^1 \to \begin{pmatrix} \text{case } M_2, ... \text{ of} \\ \quad p_1^2, ... \to A_1\ ... \\ \quad p_n^2, ... \to A_n \end{pmatrix} \end{pmatrix}$$

To save space I will sometimes define multiple nested pattern matches with the same case...of construct. See the definition of `lq` and `eq` in Appendix A.

Definition 2.19 (Multiple variable function abstraction).

$$\text{fn } x_1 : \tau, x_2 : \tau, ...\ A \quad \overset{\text{def}}{=} \quad \text{fn } x_1 : \tau.\ \text{fn } x_2 : \tau, ...\ A$$

Another space saving construct, one which stacks nested function abstractions into a single abstraction.

2.1.5 Fold

This section gives a final piece of syntactic sugar extension: $\text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_n\rangle$, referred to as the fold of data-type $\boldsymbol{T}$, and given formally in Definition 2.20. Given a data-type $\boldsymbol{T}$ with n constructors, and terms $C_1...C_2$, the function defined by $\text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_n\rangle$ takes a term of type $\boldsymbol{T}$ and, within it, replaces all constructors of $\boldsymbol{T}$ with $C_1, ..., C_n$ respectively. This section first defines the fold syntax, then gives some examples of functions in terms of fold rather than fix, finishing with some examples of the fold syntax for specific data-types.

Definition 2.20 (fold).

$$\text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_n\rangle \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f : \boldsymbol{T} \to \tau, x : \boldsymbol{T}. \\ \quad \text{case } x \text{ of} \\ \quad\quad P_1 \to C_1\ A_1^1...A_1^{m_1}\ ... \\ \quad\quad P_n \to C_n\ A_n^1...A_n^{m_n}\end{pmatrix}$$

$$\begin{aligned}\text{where}\quad & \boldsymbol{T} = (\boldsymbol{T}_1^1, ..., \boldsymbol{T}_1^{m_1}) \mid ... \mid (\boldsymbol{T}_n^1, ..., \boldsymbol{T}_n^{m_n}) \quad \text{is the definition of } \boldsymbol{T} \\ & f, x \text{ and } y_1^1, ..., y_n^{m_n} \text{ are fresh variables} \\ & P_i = \text{con}_i\langle \boldsymbol{T}\rangle\ y_i^1...y_i^{m_i} \quad \{ \text{ for all } i \le n \ \} \\ & A_i^j = \begin{cases} f\ y_j^i & \text{if } \boldsymbol{T}_j^i = \boldsymbol{T} \\ y_j^i & \text{otherwise}\end{cases} \quad \{ \text{ for all } i \le n, j \le m_i \ \}\end{aligned}$$

This syntax defines a recursive function which takes an argument of type $\boldsymbol{T}$ and returns type τ. It does this by pattern matching on the given argument, and applies the arguments of the pattern to the given C_i term for that pattern. Any recursive occurrences of the data-type within that pattern induce recursive calls to the fold function, as shown in the definition of A_i^j, representing the jth argument to the ith constructor.

The effect of this is to replace every constructor in the argument with the corresponding term C_i, where i is the index of that constructor in the recursive definition of $\boldsymbol{T}$.

Fold represents a very common form of recursion, and almost every function used in this thesis could have been defined with a fold instead of a fix. Below are some examples defined in terms of fold alongside their definition in terms of fix.

$$\texttt{add}_y = \text{fold}_{\texttt{Nat}}\langle y, \texttt{Suc}\rangle = \text{fix}\begin{pmatrix}\text{fn } f, x, y.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (f\ x'\ y)\end{pmatrix}$$

$$\texttt{snoc}_y = \text{fold}_{\texttt{List}_\tau}\langle \texttt{[}y\texttt{]}, \texttt{Cons}\rangle = \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to x :: f\ xs'\end{pmatrix}$$

$$\texttt{app}_{ys} = \text{fold}_{\texttt{List}_\tau}\langle ys, \texttt{Cons}\rangle = \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to x :: f\ xs'\end{pmatrix}$$

$$\texttt{rev} = \text{fold}_{\texttt{List}_\tau}\langle \texttt{[ ]}, \text{fn } x.\ \texttt{snoc}_x\rangle = \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[ ]} \\ \quad x :: xs' \to \texttt{snoc}_x\ (f\ xs')\end{pmatrix}$$

Indeed, the if...then...else syntax could also have been defined as a fold

$$\text{if } b \text{ then } B_1 \text{ else } B_2 \;\cong\; \text{fold}_{\texttt{Bool}}\langle B_1, B_2\rangle\, b$$

I will now give fold function for the three data-types most used in this thesis, `Nat`, $\texttt{List}_{\texttt{Nat}}$, and `Bool`.

Example 2.2 (Folding **`Nat`**). Given any type τ, and terms $C_1 : \tau$ and $C_2 : \tau \to \tau$, we have

$$\text{fold}_{\texttt{Nat}}\langle C_1, C_2\rangle = \text{fix}\begin{pmatrix}\text{fn } f : \texttt{Nat} \to \tau, x : \texttt{Nat}. & & \\ \quad\text{case } x \text{ of} & & \\ \qquad \texttt{0} & \to & C_1 \\ \qquad \texttt{Suc}\, y & \to & C_2\,(f\,y)\end{pmatrix} d$$

This function will recursively replace `0` with C_1, and `Suc` with C_2, in the number it is given as input. For example

$$\text{fold}_{\texttt{Nat}}\langle C_1, C_2\rangle\;(\texttt{Suc}\;(\texttt{Suc}\;\texttt{0})) \;\cong\; C_2\,(C_2\,C_1)$$

Example 2.3 (Folding $\texttt{List}_{\texttt{Nat}}$). Given any type τ, and terms $C_1 : \tau$ and $C_2 : \texttt{Nat} \to \tau \to \tau$, we have

$$\text{fold}_{\texttt{List}_{\texttt{Nat}}}\langle C_1, C_2\rangle = \text{fix}\begin{pmatrix}\text{fn } f : \texttt{List}_{\texttt{Nat}} \to \tau, x : \texttt{List}_{\texttt{Nat}}. & & \\ \quad\text{case } x \text{ of} & & \\ \qquad \texttt{[]} & \to & C_1 \\ \qquad y_1 :: y_2 & \to & C_2\,y_1\,(f\,y_2)\end{pmatrix}$$

This function will recursively replace `[ ]` with C_1, and `Cons` with C_2, in the list it is given as input. For example

$$\text{fold}_{\texttt{List}_{\texttt{Nat}}}\langle C_1, C_2\rangle\;(x :: y :: \texttt{[]}) \;\cong\; C_2\,x\,(C_2\,y\,C_1)$$

Example 2.4 (Folding **`Bool`**). Given any type τ, and terms $C_1 : \tau$ and $C_2 : \tau$, we have

$$\text{fold}_{\texttt{Bool}}\langle C_1, C_2\rangle = \text{fix}\begin{pmatrix}\text{fn } f : \texttt{Bool} \to \tau, x : \texttt{Bool}. & & \\ \quad\text{case } x \text{ of} & & \\ \qquad \texttt{True} & \to & C_1 \\ \qquad \texttt{False} & \to & C_2\end{pmatrix}$$

Since this fixed-point never uses its recursive call, it is equivalent to

$$\begin{array}{lll}\text{fn } x.\ \text{case } x \text{ of} & & \\ \quad \texttt{True} & \to & C_1 \\ \quad \texttt{False} & \to & C_2\end{array}$$

The above is synonymous with our if...then...else syntax, so we have

$$\text{fold}_{\texttt{Bool}}\langle C_1, C_2\rangle \cong \text{fn } b.\ \text{if } b \text{ then } C_1 \text{ else } C_2$$

2.2 Domain Theory

The previous section formalised the programming language νPCF, the terms of which this thesis is aiming to prove approximation properties for. To show that this proof system is sound, I refer to the denotational semantics of νPCF, a mapping from νPCF terms and types into mathematical objects. This section describes these mathematical objects.

(Section 2.2.1)	Domains of computation - the structure which the representation of our νPCF types must have.
(Section 2.2.2)	The specific domains that our νPCF types are mapped to. Since we have function types and data types, we need function domains and data-type domains.
(Section 2.2.3)	Least fixed-points - the representation of fix in νPCF, and a construct integral to my proof method.

2.2.1 Domains

A domain of computation, often shortened to just "domain", is a set endowed with three extra features that let it represent executable computations in a mathematical setting.

$\sqsubseteq$	A partial-order between elements (Definition 2.21).
$\bot$	A least element, such that $\bot \sqsubseteq x$ for any x. This represents the completely undefined computation (Definition 2.22).
$\bigsqcup$	Least upper-bounds (Definition 2.24) of chains (Definition 2.23), allowing us to represent recursion (Definition 2.24).

Once these three features are explained, Definition 2.25 gives the formal definition of a domain. This section then explains the notion of continuous function domains, first by describing monotone functions in Definition 2.27, then continuous functions in Definition 2.28, and finally continuous function domains in Definition 2.29. This section ends with some useful lemmas about chains.

Definition 2.21 (Partial orders $\sqsubseteq$). A binary relation $\sqsubseteq$ on a set D is a partial order iff it is:

$$\begin{array}{ll} \text{reflexive} & \forall(x \in D)\,.\, x \sqsubseteq x \\ \text{anti-symmetric} & \forall(x, y \in D)\,.\, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y \\ \text{transitive} & \forall(x, y, z \in D)\,.\, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \end{array}$$

A set D, with a partial order $\sqsubseteq$, written $(D, \sqsubseteq)$, is referred to as a partially ordered set, or poset. In the context of denotational semantics, this partial order is a "less-defined-or-equivalent" ordering, where $x \sqsubseteq y$ means that the value of x is less computed/defined than y. It is the approximation relation referred to in our introduction. Later, in Section 2.3.3 on page 37, we show that, if the denotation of one term is $\sqsubseteq$ to another, then that term also operationally approximates the other.

Definition 2.22 (Least elements $\bot$). $\bot \in D$ of poset $(D, \sqsubseteq)$ is a least element if

$$\forall (x \in D) \,.\, \bot \sqsubseteq x$$

This element $\bot$ is generally referred to as bottom, but in the context of computation we refer to it as undefined. It represents a computation with no result, such as a non-terminating computation, or an incomplete pattern match.

Definition 2.23 (Chains). Given a poset $(D, \sqsubseteq)$, a chain is any infinite sequence of elements $d_1, d_2, ... \in D$ such that $d_1 \sqsubseteq d_2 \sqsubseteq ....$

Definition 2.24 (Least upper-bounds of chains $\bigsqcup_i d_i$). Given a poset $(D, \sqsubseteq)$, and a chain $d_1 \sqsubseteq d_2 \sqsubseteq ... \in D$. An upper-bound of the chain d_i is any element d' s.t. $\forall i \,.\, d_i \sqsubseteq d'$. A least upper-bound, if it exists, is the unique smallest such element by $\sqsubseteq$, and is written

$$\bigsqcup_i d_i \qquad \text{the least upper-bound of the chain } d_1 \sqsubseteq d_2 \sqsubseteq ...$$

By definition, a least upper-bound of the chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$ is the unique element $\bigsqcup_i d_i$ such that

$$\begin{array}{lll} \forall i \,.\, d_i \sqsubseteq \bigsqcup_i d_i & & \text{it is an upper-bound} \\ \forall (d' \in D) \,.\, (\forall i \,.\, d_i \sqsubseteq d') \quad \Rightarrow & \bigsqcup_i d_i \sqsubseteq d' & \text{it is least} \end{array}$$

Definition 2.25 (Domain of computation). A poset $(D, \sqsubseteq)$ is a domain of computation if it has a least element $(\bot)$ and if every chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$ has a least upper-bound $\bigsqcup_i d_i$.

Definition 2.26 (Chain-closed predicates). A predicate $\mathcal{P}$ over a domain $(D, \sqsubseteq)$ is called chain-closed if, for any chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$ such that $\forall i \,.\, \mathcal{P}(d_i)$, we have $\mathcal{P}(\bigsqcup_i d_i)$.

It is common to see the term admissible in the literature, referring to a chain-closed predicate $\mathcal{P}$ for which $\mathcal{P}(\bot)$ also holds. This term often occurs in induction rules, where $\mathcal{P}(\bot)$ refers to the base case of this induction. The reason I do not use the term admissible in this thesis is that I find that explicitly stating the $\mathcal{P}(\bot)$ base case makes the inductive shape of the proof more obvious and intuitive to the reader.

Definition 2.27 (Monotone functions). Given domains $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, a function $f : D \to E$ is called monotone if

$$\forall (d, d' \in D) \,.\, d \sqsubseteq_D d' \Rightarrow f\ d \sqsubseteq_E f\ d'$$

Definition 2.28 (Continuous functions). Given domains $(D, \sqsubseteq)$ and $(E, ...)$, a function $f : D \to E$ is called continuous if it is monotone and

$$\forall (d_1 \sqsubseteq d_2 \sqsubseteq ...) \;.\; f\left(\bigsqcup_i d_i\right) = \bigsqcup_i (f\; d_i)$$

The above property is referred to as "preserves upper bounds".

Definition 2.29 (Continuous function domains). Given domain $(D, \sqsubseteq_D)$, and domain $(E, \sqsubseteq_E)$ with bottom element $\bot_E$, the domain of continuous functions from D to E is denoted $(D \to E, \sqsubseteq_{D\to E})$. In the context of domain theory, $D \to E$ refers to only those functions from D to E which are continuous, which is to say, all functions over domains used in this thesis are continuous.

The ordering $\sqsubseteq_{D\to E}$ is defined as

$$f \sqsubseteq_{D\to E} g \;\overset{\text{def}}{\Leftrightarrow}\; \forall (d \in D) \;.\; f\; d \sqsubseteq_E g\; d$$

The bottom element $\bot_{D\to E}$ is defined as

$$\bot_{D\to E} \;\overset{\text{def}}{=}\; \lambda (d \in D) \;.\; \bot_E$$

Lemma 2.1. For any chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$ in some domain, for all $n \in \mathbb{N}$

$$\bigsqcup_i d_i = \bigsqcup_i d_{n+i}$$

Proof. The chain $d_{n+1} \sqsubseteq d_{n+2} \sqsubseteq ...$ is just the chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$ with a finite portion chopped off the beginning, which will not affect the upper-bound of the chain. $\square$

Lemma 2.2. For any two chains $d_1 \sqsubseteq d_2 \sqsubseteq ...$ and $d'_1 \sqsubseteq d'_2 \sqsubseteq ...$ both in the same domain

$$\begin{array}{ll} \text{if} & \forall (i \in \mathbb{N}) \;.\; d_i \sqsubseteq d'_i \\ \text{then} & \bigsqcup_i d_i \sqsubseteq \bigsqcup_i d'_i \end{array}$$

Proof. Showing $\bigsqcup_i d'_i$ to be an upper-bound of the chain $d_1 \sqsubseteq d_2 \sqsubseteq ...$, gives us our goal, since $\bigsqcup_i d_i$ is the least upper-bound of the same chain. As $\bigsqcup_i d'_i$ is an upper-bound of the chain $d'_1 \sqsubseteq d'_2 \sqsubseteq ...$ we have $\forall i \;.\; d'_i \sqsubseteq \bigsqcup_i d'_i$, by transitivity and our antecedent, we have that $\forall i \;.\; d_i \sqsubseteq \bigsqcup_i d'_i$, hence $\bigsqcup_i d'_i$ is an upper bound of the chain $d_1 \sqsubseteq d_2 \sqsubseteq ....$ $\square$

2.2.2 νPCF domains

The previous section describes the structure a domain must have in order to represent executable computation. It also describes how, given domains D and E, we can build a continuous function domain $D \to E$, but we have seen no base domains out of which to build these function domains.

The base types in νPCF are data-types, and in this section I describe the domains which these data-types will be mapped to. Data-type domains require three preliminary concepts: lifting, products, and coproducts. This section finishes with some example data-type domains, which represent the νPCF data-types `Nat` and $\texttt{List}_{\texttt{Nat}}$.

Definition 2.30 (Lifted poset). Given poset $(D, \sqsubseteq)$, the lift of this poset is denoted $(D_\bot, \sqsubseteq_\bot$ $)$, where $D_\bot$ is the set D with a single element $\bot$ added. The ordering $\sqsubseteq_\bot$ is the ordering $\sqsubseteq$ extended such that $\bot$ is $\sqsubseteq_\bot$ to every element in D, which is to say

$$\sqsubseteq_\bot \;\stackrel{\text{def}}{=}\; \sqsubseteq \;\cup \{\; (\bot, d) \mid d \in D_\bot \;\}$$

Definition 2.31 (Poset product). Given posets $(D_1, \sqsubseteq_1)$ ... $(D_n, \sqsubseteq_n)$, the product of these posets is denoted $(\prod_i D_i, \sqsubseteq_\times)$, where

$$\textstyle\prod_i D_i \;\stackrel{\text{def}}{=}\; \{\; (d_1, ..., d_n) \mid d_1 \in D_1, ..., d_n \in D_n \;\}$$

$$(d_1, ..., d_n) \sqsubseteq_\times (d'_1, ..., d'_n) \;\stackrel{\text{def}}{\Leftrightarrow}\; d_1 \sqsubseteq_1 d'_1 \wedge \;...\; \wedge d_n \sqsubseteq_n d'_n$$

We can also write $D_1 \times ... \times D_n$ to mean $\prod_i D_i$. The product of zero poset is the single element poset $(\{()\}, \{((), ())\})$.

Definition 2.32 (Poset coproduct). Given posets $(D_1, \sqsubseteq_1)$ ... $(D_n, \sqsubseteq_n)$, the coproduct of these posets is denoted $(\coprod_i D_i, \sqsubseteq_+)$, where

$$\textstyle\coprod_i D_i \;\stackrel{\text{def}}{=}\; \bigcup_i \{\; \mathsf{inj}_i\, d \mid d \in D_i \;\}$$

$$\mathsf{inj}_i\, d \sqsubseteq_+ \mathsf{inj}_j\, d' \;\stackrel{\text{def}}{\Leftrightarrow}\; i = j \wedge d \sqsubseteq_i d'$$

The notation inj_i is referred to as the ith injector of $\coprod_j D_j$. We can also write $D_1 + ... + D_n$ to mean $\coprod_i D_i$. The coproduct of zero posets is the empty poset $(\emptyset, \emptyset)$.

Definition 2.33 (Data-type domains). The domains we use to denote our νPCF data-types are defined as the solution to a set of recursive domain equations, each of the form

$$D \cong \left(\textstyle\coprod_i \left(\prod_j D_i^j\right)\right)_\bot$$

This is to say a lifted coproduct of products of a family of domains D_i^j where all of D_i^j are either a recursive occurrence of the D we are defining, or another domain defined in the same manner. That such an equation has a solution, and that this solution is itself a domain, is not covered in this thesis and we instead refer the reader to the domain theory text by Abramsky and Jung [1].

Example 2.5 (Domain of non-strict natural numbers nat).

$$\mathsf{nat} \cong () + (\mathsf{nat})$$

The solution, nat, to the above domain equation is the domain of non-strict natural numbers. For this domain we can then define the zero element, $\mathsf{zero} \in \mathsf{nat}$, and the successor operator, $\mathsf{suc} : \mathsf{nat} \to \mathsf{nat}$, as

$$\begin{aligned} \mathsf{zero} &= \mathsf{inj}_1\,() \\ \mathsf{suc} &= \lambda(x \in \mathsf{nat}) \,.\, \mathsf{inj}_2\,(x) \end{aligned}$$

This domain is what the νPCF data-type `Nat` (Definition 2.2) is mapped to.

Example 2.6 (Domain of non-strict lists of nat).

$$\mathsf{list\ nat} \cong () + (\mathsf{nat} \times \mathsf{list\ nat})$$

If nat is the domain of non-strict natural numbers, then the solution, $\mathsf{list\ nat}$, to the above domain equation is the domain of non-strict lists of non-strict natural numbers. For this domain, we can then define the empty list, $\mathsf{nil} \in \mathsf{list\ nat}$, and the cons operator, $\mathsf{cons} : \mathsf{nat} \to \mathsf{list\ nat} \to \mathsf{list\ nat}$, as

$$\begin{aligned} \mathsf{nil} &= \mathsf{inj}_1\,() \\ \mathsf{cons} &= \lambda(x \in \mathsf{nat})(xs \in \mathsf{list\ nat}) \,.\, \mathsf{inj}_2\,(x, xs) \end{aligned}$$

This domain is what the νPCF data-type $\mathtt{List}_{\mathtt{Nat}}$ (Definition 2.2) is mapped to.

2.2.3 Least fixed-points

Least fixed-points allow us to represent the fix construct from νPCF as a mathematical object and, in general, are used to model recursion, or looping, in denotational semantics. This section first defines what a pre-fixed-point, post-fixed-point and fixed-point are (Definition 2.34). Then it describes the least fixed-point construct $\mathsf{fix}\,(f)$.

It's all very well to call $\mathsf{fix}\,(f)$ a least fixed-point, but I back-up this name by showing it is a fixed-point in Lemma 2.3 and that it is the least pre-fixed-point, and hence the least fixed-point in Lemma 2.4. Furthermore, Lemma 2.4 is the main proof rule in our theorem prover (see page 103).

Definition 2.34 (Fixed-points). Given a domain D and a function on that domain $f : D \to D$, $d \in D$ is called a a pre-fixed-point of f if

$$f\,d \sqsubseteq d$$

$d \in D$ is a post-fixed-point of f if

$$d \sqsubseteq f\,d$$

$d \in D$ is a fixed-point of f if it is both a post-fixed-point and pre-fixed-point, which is to say that

$$f\ d = d$$

Definition 2.35 (Least fixed-point). Given domain D and a continuous function $f : D \to D$, a unique least fixed-point of f exists and is given by

$$\mathsf{fix}\,(f) \ \overset{\text{def}}{=} \ \bigsqcup_i (f^i \bot)$$

The notation f^i means f composed with itself i-many times, so $f^0 = id$ and $f^{n+1} = f \circ f^n$. As D is a domain, and $\bot \sqsubseteq f\ \bot \sqsubseteq f^2\ \bot \sqsubseteq ...$ is an increasing chain, by completeness we know that this object $\mathsf{fix}\,(f)$ exists.

Lemma 2.3 (Least fixed-points are fixed-points). Fix a domain D and continuous function $f : D \to D$, then $\mathsf{fix}\,(f) = f\ (\mathsf{fix}\,(f))$.

Proof.

$f\ (\mathsf{fix}\,(f))$

$\quad = \{$ by definition of $\mathsf{fix}\,(f)$ $\}$

$f\ (\bigsqcup_i f^i \bot)$

$\quad = \{$ by continuity of f $\}$

$\bigsqcup_i f^{i+1} \bot$

$\quad = \{$ by Lemma 2.1 $\}$

$\bigsqcup_i f^i \bot$

$\quad = \{$ by definition of $\mathsf{fix}\,(f)$ $\}$

$\mathsf{fix}\,(f)$

□

Lemma 2.4 (Least (pre-)fixed-point principle). This lemma is the cornerstone of my fixed-point promotion proof technique, outlined later in Chapter 3, which is how my automated prover Elea proves properties without using induction or co-induction.

Fix a domain D, a $d \in D$ and a continuous $f : D \to D$.

$$\begin{array}{ll} \text{if} & f\ d \sqsubseteq d \\ \text{then} & \mathsf{fix}\,(f) \sqsubseteq d \end{array}$$

Proof. By induction over $n \in \mathbb{N}$, we can show that $\forall(n \in \mathbb{N}) \,.\, f^m\bot \sqsubseteq d$. The base case $\bot \sqsubseteq d$ holds trivially, and the inductive case $f^n\bot \sqsubseteq d \Rightarrow f(f^n\bot) \sqsubseteq d$ holds by

$$
\begin{aligned}
& f^n\bot \sqsubseteq d \\
& \quad \Rightarrow \{ \text{ by monotonicity of } f \text{ } \} \\
& f(f^n\bot) \sqsubseteq f\ d \\
& \quad \Rightarrow \{ \text{ by transitivity of } (\sqsubseteq) \text{ with } f\ d \sqsubseteq d \text{ } \} \\
& f(f^n\bot) \sqsubseteq d
\end{aligned}
$$

From $\forall(n \in \mathbb{N}) \,.\, f^n\bot \sqsubseteq d$ we know that d is an upper-bound of the chain $\bot \sqsubseteq f\ \bot \sqsubseteq f^2\bot \sqsubseteq \ldots$ and, as $\mathsf{fix}\,(f)$ is defined to be the least upper-bound of this chain, we know $\mathsf{fix}\,(f) \sqsubseteq d$. □

Corollary 2.1 (Least fixed-points are least). Fix a domain D, a $d : D$ and a continuous function $f : D \to D$. If d is a fixed-point of f, then $\mathsf{fix}\,(f) \sqsubseteq d$.

Proof. Lemma 2.4 as $f\ d = d \;\Rightarrow\; f\ d \sqsubseteq d$ □

Lemma 2.5 (Truncation induction). This lemma provides the mechanism for the proof of soundness for the rewrite system I describe in this thesis. In particular, for the fusion rewrite steps given in Chapter 5. This lemma is due to Morris [53], and can also be extended to strong induction if required [50], as given in Lemma 2.6, though I did not require this strengthening within this thesis.

Let $(D, \sqsubseteq)$ be a domain, $\mathcal{P}$ be a chain-closed predicate over D, and $f : D \to D$ be a continuous function.

$$
\begin{array}{ll}
\text{if} & \mathcal{P}(\bot) \\
\text{and} & \forall(n \in \mathbb{N}) \,.\, \mathcal{P}(f^n\bot) \Rightarrow \mathcal{P}(f^{n+1}\,\bot) \\
\text{then} & \mathcal{P}(\mathsf{fix}\,(f))
\end{array}
$$

Proof. Using induction over $n \in \mathbb{N}$, where the first antecedent is the base case and the second the inductive case, gives us $\forall(n \in \mathbb{N}) \,.\, \mathcal{P}(f^n\bot)$. Then the definition of chain-closedness (Definition 2.26) and that $\bot \sqsubseteq f\ \bot \sqsubseteq f^2\bot \sqsubseteq \ldots$ is a chain gives us our goal. □

Lemma 2.6 (Strong truncation induction). Let $(D, \sqsubseteq)$ be a domain, $\mathcal{P}$ be a chain-closed predicate over D, and $f : D \to D$ be a continuous function.

$$
\begin{array}{ll}
\text{if} & \forall(n \in \mathbb{N}) \,.\, (\forall(m < n) \,.\, \mathcal{P}(f^m\bot)) \Rightarrow \mathcal{P}(f^n\,\bot) \\
\text{then} & \mathcal{P}(\mathsf{fix}\,(f))
\end{array}
$$

Proof. By strong induction, the antecedent gives us that $\forall (n \in \mathbb{N}) \mathrel{.} \mathcal{P}(f^n \bot)$, then chain-closedness of $\mathcal{P}$ gives us our goal. □

Lemma 2.7 (Fixed-point induction). Fixed-point induction is a proof method which is strictly weaker than truncation induction, but which occurs much more within the literature. While it is often sufficient to prove many properties, there are some which it cannot, see Remark 2.1.

Let D be a domain, $\mathcal{P}$ be a chain-closed predicate over D, and $f : D \to D$ be a continuous function.

$$\begin{array}{ll} \text{if} & \mathcal{P}(\bot) \\ \text{and} & \forall (d \in D) \mathrel{.} \mathcal{P}(d) \Rightarrow \mathcal{P}(f\ d) \\ \text{then} & \mathcal{P}(\mathsf{fix}\,(f)) \end{array}$$

Proof. By truncation induction using $d = f^n \bot$. □

Remark 2.1 (Truncation induction vs. fixed-point induction). I believe truncation induction is a very powerful method, which has been largely overlooked by the theorem proving community. For example, Gibbons' comparison between proof methods for corecursive programs [24], which includes fixed-point induction, does not include truncation induction, even though it can prove strictly more properties than fixed-point induction.

To give an example of a property truncation induction can prove which fixed-point induction cannot, take the example given in the introduction: $\texttt{add}\ x\ x \sqsubseteq \texttt{double}\ x$. In general, any proof which requires some property of the unrolled function will not be provable by fixed-point induction, since this property has been generalised away. In truncation induction, however, we generalise the function to some finite unrolling, which preserves almost every property of the original function.

Lemma 2.8 (Fixed-point fusion). Fixed-point fusion is a simple method for rewriting terms by fusing contexts into fixed-points. It is a strictly less applicable technique than the truncation fusion technique I will detail in Chapter 5. There have been multiple existing rewrite systems based upon fixed-point fusion [54, 73, 56].

Let D and E be domains and $f : D \to E$, $g : D \to D$ and $h : E \to E$ be continuous functions.

$$\begin{array}{ll} \text{if} & f\ \bot = \bot \\ \text{and} & \forall (d \in D, e \in E) \mathrel{.} f\ d \sqsubseteq e \Rightarrow f\ (g\ d) \sqsubseteq h\ e \\ \text{then} & f\ (\mathsf{fix}\,(g)) \sqsubseteq \mathsf{fix}\,(h) \end{array}$$

Proof. By fixed-point induction on chain-closed predicate $\mathcal{P}(d) \overset{\text{def}}{\Leftrightarrow} f\ d \sqsubseteq \mathsf{fix}\,(h)$ □

An equivalent and more common statement of this lemma is given below.

$$\begin{array}{ll} \text{if} & f\perp = \perp \\ \text{and} & f \circ g \sqsubseteq h \circ f \\ \text{then} & f\left(\mathsf{fix}\left(g\right)\right) \sqsubseteq \mathsf{fix}\left(h\right) \end{array}$$

2.3 Denotational semantics of νPCF

Now that the previous section has given an introduction to the underlying domain theory, this section describes the denotational semantics of νPCF.

First, Section 2.3.1 gives a mapping from νPCF types to domains, written $[\![\tau]\!]$. Section 2.3.2 then gives a mapping from νPCF terms to continuous functions, written $[\![A]\!]$. Even non-function terms are denoted by functions, as they must always take an argument which provides the values of their free variables. This value environment, ranged over by ρ and η, is a mapping from the free variables of a term to objects in the domain given by their type. Without a value environment, we would have nothing to represent the free variables of a term when we give its denotational semantics.

As explained in the introduction, the overall aim of this thesis is to prove properties of operational approximation between terms. To use denotational semantics to prove this method sound means that we need to relate denotation to operation. To this end, Section 2.3.3 gives a formal definition of operational approximation and then shows that if the denotation of A approximates that of B, i.e. $[\![A]\!] \sqsubseteq [\![B]\!]$, then A operationally approximates B, i.e. $A \precsim B$.

2.3.1 Denoting νPCF types

Definition 2.36 (Denoting νPCF data-types). Data-types in νPCF are represented as a set of potentially recursive equations:

$$\boldsymbol{T} = (\boldsymbol{T}_1^1, ..., \boldsymbol{T}_1^{n_1}) \mid ... \mid (\boldsymbol{T}_m^1, ..., \boldsymbol{T}_m^{n_m})$$

For every such definition, I define a corresponding recursive domain equation on $\hat{\boldsymbol{T}}$, as explained in Definition 2.33 on page 30.

$$\hat{\boldsymbol{T}} \cong \left(\coprod_i \left(\prod_j \hat{\boldsymbol{T}}_i^j\right)\right)_\perp$$

$\hat{\boldsymbol{T}}$ is the domain which represents the νPCF data-type $\boldsymbol{T}$.

Definition 2.37 (Denoting νPCF types).

$$\begin{aligned} [\![\tau \to \tau']\!] &\stackrel{\text{def}}{=} [\![\tau]\!] \to [\![\tau']\!] \\ [\![\boldsymbol{T}]\!] &\stackrel{\text{def}}{=} \hat{\boldsymbol{T}} \end{aligned}$$

$[\![\tau]\!]$ returns the domain associated with the given type. Function types are represented as function domains and data-types as given in Definition 2.36.

2.3.2 Denoting νPCF terms

This section gives the mapping from νPCF terms into mathematical objects. For any term A and type environment Γ, if we have $\Gamma \vdash A : \tau$, then the denotational semantics of A, written $[\![A]\!]$, is a continuous function

$$[\![A]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$

The argument to this function, an element of $[\![\Gamma]\!]$, is referred to as a value environment, and represents a mapping from the free variables of A to objects in the domain given by their type. Without this value environment, we would have no way of representing the free variables of a term.

Definition 2.38 gives the domain of value environments, $[\![\Gamma]\!]$. Definition 2.39 gives the above νPCF term mapping, $[\![A]\!]$. In this thesis I use the notation $A \sqsubseteq B$ to mean $[\![A]\!] \sqsubseteq [\![B]\!]$, which is to say denotational approximation between terms, and the notation $A \equiv B$ to mean $[\![A]\!] = [\![B]\!]$, which is denotational equivalence.

Definition 2.38 (Value environments $\rho, \eta \in [\![\Gamma]\!]$).

$$[\![\Gamma]\!] \stackrel{\text{def}}{=} (x \in \mathsf{dom}\,(\Gamma)) \to [\![\Gamma(x)]\!]$$

Recall that Γ is a type environment, mapping the free variables of a term to their types. A value environment is an element of $[\![\Gamma]\!]$, ranged over by ρ and η, and is a function mapping the same free variables to objects of the domain denoted by their type. The notation used here is dependent function notation, wherein the return type changes depending upon the value passed in as the argument.

Definition 2.39 (Denoting νPCF terms).

$$\llbracket x \rrbracket\, \rho \overset{\text{def}}{=} \rho\, x$$

$$\llbracket F\ A \rrbracket\, \rho \overset{\text{def}}{=} \llbracket F \rrbracket\, \rho\, (\llbracket A \rrbracket\, \rho)$$

$$\llbracket \text{fn } x : \tau .\, A \rrbracket\, \rho \overset{\text{def}}{=} \lambda(d \in \llbracket \tau \rrbracket)\ .\ \llbracket A \rrbracket\, \rho[x \mapsto d]$$

$$\llbracket \text{fix}\, (F) \rrbracket\, \rho \overset{\text{def}}{=} \mathsf{fix}\, (\llbracket F \rrbracket\, \rho)$$

$$\llbracket \text{fix}^a\, (F) \rrbracket\, \rho \overset{\text{def}}{=} (\llbracket F \rrbracket\, \rho)^a \bot$$

$$\llbracket \text{con}_i \langle \boldsymbol{T} \rangle \rrbracket\, \rho \overset{\text{def}}{=} \lambda(d_1 \in \llbracket \tau_n \rrbracket)\ ...\ (d_n \in \llbracket \tau_n \rrbracket)\ .\ \mathsf{inj}_i\, (d_1, ..., d_n) \quad \text{where} \quad (\tau_1, ..., \tau_n) = \mathsf{conArgs}_i\, \boldsymbol{T}$$

$$\left\llbracket \text{case } A \text{ of } P_1 \to B_1\ ...\ P_m \to B_m \right\rrbracket\, \rho \overset{\text{def}}{=} \bot$$
if $\bot = \llbracket A \rrbracket\, \rho$
or $\mathsf{inj}_i\, (...) = \llbracket A \rrbracket\, \rho$
and $(P_1, ..., P_n) \notin \mathsf{dom}\, (\mathsf{findPattern}_i)$ (no pattern matches A)

$$\left\llbracket \text{case } A \text{ of } P_1 \to B_1\ ...\ P_m \to B_m \right\rrbracket\, \rho \overset{\text{def}}{=} \llbracket B_i \rrbracket\, (\rho[x_1 \mapsto d_1]...[x_n \mapsto d_n])$$
where $\mathsf{inj}_j\, (d_1, ..., d_n) = \llbracket A \rrbracket\, \rho$
$K\ x_1...x_n = P_i = \mathsf{findPattern}_j(P_1...P_n)$

Given $\Gamma \vdash A : \tau$, the denotation of a term A, written $\llbracket A \rrbracket$, is a continuous function from a value environment $\rho \in \llbracket \Gamma \rrbracket$ to the domain $\llbracket \tau \rrbracket$. I do not prove that this denotation is continuous, but I believe it follows easily from the continuity of the denotation of Plotkin's original PCF [58].

2.3.3 Relating denotation to operation

The overall aim of this thesis is to build a theorem prover for properties of operational approximation between νPCF terms. I prove my method sound by showing that, if my tool proves the property "$A \sqsubseteq B$", where $A \sqsubseteq B$ is a term in the property language of Elea, then we have $A \underset{\sim}{\sqsubset} B$, i.e. a proof of denotational approximation within my tool gives us a proof of observational approximation at the meta level.

This section formally defines observational approximation $\underset{\sim}{\sqsubset}$ (Definition 2.40) and then gives Theorem 1, which states that denotational approximation implies observational approximation. In Chapter 9, I show that my system proves denotational approximation, hence it proves observational approximation.

Theorem 1 is proven using two lemmas, soundness and adequacy, which are stated without proof as Lemma 2.9 and Lemma 2.10. Soundness states that denotation is invariant under evaluation. Adequacy states that if a term is denotationally approximated by a value, then evaluation of that term will terminate with a value. The contrapositive of adequacy states that, if a term is denotationally non-terminating, then it will be operationally non-terminating, a complementary property to soundness.

Definition 2.40 (Observational approximation).

$$A \precsim B \overset{\text{def}}{\Leftrightarrow} \forall \mathcal{C} \,.\, \mathcal{C}[A] \Downarrow \;\Rightarrow\; \mathcal{C}[B] \Downarrow$$

A observationally approximates B, written $A \precsim B$, if we can replace any instance of A with B in a terminating program without affecting its observable behaviour.

Lemma 2.9 (Soundness). Given term $A : \tau$ and value $V : \tau$

$$\text{if} \quad A \Downarrow_\tau V \qquad \text{then} \quad [\![A]\!] = [\![V]\!]$$

Lemma 2.10 (Adequacy). Given term $A : \tau$ and value $V : \tau$

$$\text{if} \quad [\![V]\!] \sqsubseteq [\![A]\!] \qquad \text{then} \quad A \Downarrow$$

Theorem 1. Given terms A and B:

$$\text{if} \quad [\![A]\!] \sqsubseteq [\![B]\!] \qquad \text{then} \quad A \precsim B$$

Proof. Fixing some $\mathcal{C}$, we can show that $\mathcal{C}[A] \Downarrow \Rightarrow \mathcal{C}[B] \Downarrow$, as per the definition of $\precsim$

$\mathcal{C}[A] \Downarrow$
  $\Leftrightarrow$ { by definition of $\Downarrow$ }
$\exists V \,.\, \mathcal{C}[A] \Downarrow_\tau V$
  $\Rightarrow$ { by soundness }
$\exists V \,.\, [\![\mathcal{C}[A]]\!] = [\![V]\!]$
  $\Rightarrow$ { by monotonicity of term denotations and $[\![A]\!] \sqsubseteq [\![B]\!]$ }
$\exists V \,.\, [\![V]\!] \sqsubseteq [\![\mathcal{C}[B]]\!]$
  $\Rightarrow$ { by adequacy }
$\mathcal{C}[B] \Downarrow$

□

2.4 Term rewriting and termination

The theorem proving power of Elea comes entirely from its term rewriting system. This section gives an introduction to term rewriting systems, including the method I have used to show Elea terminates (Section 2.4.1), which is called the homeomorphic embedding. The fusion rewrite rules given in Chapter 5 are a type of unfold-fold rewrite rule, which are explained in Section 2.4.2.

Definition 2.41 (Term rewriting rules $L \longrightarrow R$). A term rewriting system is a set of rules of the form $L \longrightarrow R$, where L and R are both terms and the free variables of R are a subset of those of L. The rule $L \longrightarrow R$ denotes that we can rewrite any instance of the term L to the term R.

Definition 2.42 (Transitive closure $L \longrightarrow_+ R$).

$$\frac{A \longrightarrow B}{A \longrightarrow_+ B} \qquad\qquad \frac{A \longrightarrow B \qquad B \longrightarrow_+ C}{A \longrightarrow_+ C}$$

The transitive closure of a set of rewrite rules is denoted $L \longrightarrow_+ R$ and means that L can be rewritten to R in one or more applications of any rewrite rules in our set.

2.4.1 Proving rewrite systems terminate

This section is a handy guide on turning any term rewriting system (Definition 2.41) into a terminating one using the theory of well-quasi orders, particularly the well-quasi-order known as the homeomorphic embedding. This is based on the work of Michael Leuschel [47].

If we have a well-quasi-order (Definition 2.44) on terms, we can use its whistle (Definition 2.45) to construct the well-founded transitive closure (Definition 2.46) of a set of rewrite rules which, unlike the regular transitive closure (Definition 2.42), is guaranteed to be terminating (Lemma 2.12).

This section goes on to define a specific well-quasi-order on terms, the homeomorphic embedding (Section 2.4.1.1), which is experimentally very effective at controlling termination (Remark 2.2). This is the ordering used by the original version of Klyuchnikov's supercompiler HOSC [41], which has since been extended [42], but I did not find this extension was necessary for controlling termination in Elea, so it has not been included.

Definition 2.43 (Quasi-order). A relation $(\leq)$ is a quasi-order (or pre-order) if it is

reflexive	$\forall x \,.\, x \leq x$
transitive	$\forall x, y, z \,.\, x \leq y \wedge y \leq z \Rightarrow x \leq x$

Definition 2.44 (Well-quasi-order). A quasi-order, $\leq$, is a well-quasi-order if for any infinite sequence of elements $A_1, A_2, ...$ there will always be an $i < j$ s.t. $A_i \leq A_j$.

Definition 2.45 (Whistle of $\leq$).

$$\mathcal{W}_{\leq} \overset{\text{def}}{=} \{ A_1, A_2, ... \mid \exists (i < j) \,.\, A_i \leq A_j \}$$

Given any binary relation $\leq$ the whistle of that relation $\mathcal{W}_{\leq}$ is defined as the set of all sequences in which there exists an object which is $\leq$ to an object later in the sequence.

Lemma 2.11. If $\leq$ is a well-quasi-order then $\mathcal{W}_{\leq}$ contains all infinite sequences.

Proof. Fix an arbitrary infinite sequence $A_1, A_2, ...$, by the definition of a well-quasi-order we have $(\exists (i < j) \,.\, A_i \leq A_j)$, so $A_1, A_2, ... \in \mathcal{W}_{\leq}$. $\square$

Definition 2.46 (Well-founded transitive closure).

$$\frac{A \longrightarrow B}{\mathcal{H} \vdash A \longrightarrow_+ B} \text{ if } \mathcal{H} \notin \mathcal{W}_{\leq} \qquad \frac{A \longrightarrow B \qquad \mathcal{H}, A \vdash B \longrightarrow_+ C}{\mathcal{H} \vdash A \longrightarrow_+ C} \text{ if } \mathcal{H} \notin \mathcal{W}_{\leq}$$

Given a well-quasi-order on terms $\leq$, the well-founded transitive closure of a set of rewrite rules is given inductively from these two derivations, where $\mathcal{H}$ is a sequence of terms referred to as the term history.

Lemma 2.12 (The well-founded transitive closure of a rewrite system always terminates). Given a well-quasi-order $\leq$, the derivation rules from Definition 2.46 are well-founded. This is to say that there is no infinite sequence of $A_1, A_2, ...$ such that

$$\cfrac{A_1 \longrightarrow A_2 \qquad \cfrac{A_2 \longrightarrow A_3 \qquad \cfrac{\vdots}{\{A_1, A_2\} \vdash A_3 \longrightarrow_+ ...}}{\{A_1\} \vdash A_2 \longrightarrow_+ ...}}{\emptyset \vdash A_1 \longrightarrow_+ ...}$$

Proof. Assume that we have such an infinite sequence, by Definition 2.46 we have $A_1, A_2, ... \notin \mathcal{W}_{\leq}$, which contradicts Lemma 2.11. $\square$

2.4.1.1 The homeomorphic embedding

Definition 2.46 gave a terminating transitive closure of a rewrite system which assumed some well-quasi-order on terms. This section gives the specific wqo that is used in this thesis: the homeomorphic embedding.

Remark 2.2 discusses what it means for a well-quasi-order to be effective at controlling termination and explains that the homeomorphic embedding is such an ordering. Then, Definition 2.47 gives the homeomorphic embedding ordering $\trianglelefteq$ and Lemma 2.13 proves this ordering to be well-founded, and hence that it can act as a termination ordering for a rewriting system.

Using the whistle of the homeomorphic embedding, in Definition 2.48 I define a ordering on sets of terms $\prec$, which is used later in this thesis to prove termination. I prove this ordering to be well-founded in Lemma 2.14.

Remark 2.2 (What is an effective wqo for termination?). As demonstrated by Lemma 2.12, we can use a well-quasi-order to control termination of any process which produces a sequence of elements $A_1, A_2, ...$ by ensuring that all such sequences are finite. In the case of rewriting systems, this sequence of elements is a sequence of terms, where $A_i \longrightarrow A_{i+1}$, and our wqo ensures that there is no infinite sequence of rewrites.

An ”effective” wqo $\leq$ is one which allows for all the rewrites you want your system to perform, which means that for any term A we want to (transitively) rewrite to term B, we need $A \not\leq B$. Choosing a wqo to control termination of a rewrite system is motivated purely by experimental evidence. There is no formal reason, of which I am aware, why the homeomorphic embedding is the well-quasi-order of choice for the discerning term rewriter, but experimentally it performs very well. In developing Elea, I did not find a single instance of a desired rewrite which the homeomorphic embedding prevented.

The only disadvantage of the homeomorphic embedding is that it is computationally expensive to check. This is not an issue for theorem proving, but it is for this reason Bolingbroke’s program optimising supercompiler [6] uses a faster method called tag bags [7]. I did not use this method for Elea, as I found it blocked some desirable rewrites, and hence was not an effective wqo for my purposes.

Definition 2.47 (Homeomorphic embedding $\trianglelefteq$). Given a grammar of expressions ranged over by E, where its node labels are ranged over by φ, and its variables by x

$$E \ ::= \ x \ | \ \varphi(E_1, ..., E_n)$$

The homeomorphic embedding on terms in this grammar is inductively defined by

$$\frac{}{x \trianglelefteq x'} \qquad \frac{\exists(i \leq n) \,.\, E \trianglelefteq E_i}{E \trianglelefteq \varphi(E_1, ..., E_n)} \qquad \frac{\forall(i \leq n) \,.\, E_i \trianglelefteq E'_i}{\varphi(E_1, ..., E_n) \trianglelefteq \varphi(E'_1, ..., E'_n)}$$

Lemma 2.13 ($\trianglelefteq$ is a well-quasi-order on terms). If an expression grammar has a finite set of node labels, then $\trianglelefteq$ is a well-quasi-order on expressions in that grammar. The proof of this property follows from Kruskal's tree embedding theorem, and is detailed in the paper "Homeomorphic Embedding for Online Termination of Symbolic Methods" [47].

Definition 2.48 (Well-founded ordering on term histories). Let $\mathcal{H}$ and $\mathcal{H}'$ be sequences of terms, using the whistle ($\mathcal{W}$) of the homeomorphic embedding ($\trianglelefteq$) we can define a well-founded ordering ($\triangleright$) on term histories:

$$\mathcal{H} \triangleright \mathcal{H}' \quad \Leftrightarrow \quad \mathcal{H}' \notin \mathcal{W}_{\trianglelefteq} \wedge \exists \mathcal{H}'' \,.\, \mathcal{H}' = \mathcal{H}, \mathcal{H}''$$

This is an ordering that will form part of the proof that my Elea tool is terminating.

Lemma 2.14 ($\triangleright$ on term histories is well-founded). There exists no infinite chain of term sequences $\mathcal{H}_1 \triangleright \mathcal{H}_2 \triangleright ...$, where $\triangleright$ is the ordering on term sequences given in Definition 2.48.

Proof. Assume such an infinite chain $\mathcal{H}_1 \triangleright \mathcal{H}_2 \triangleright ...$ exists. Fix the infinite sequence $A_1, A_2, ...$ as the least sequence of which every $\mathcal{H}_i$ is a sub-sequence. We have that $A_1, A_2, ... \in \mathcal{W}_{\trianglelefteq}$, as $\mathcal{W}_{\trianglelefteq}$ contains every infinite sequence (Lemma 2.11). From the definition of $\mathcal{W}$ we have some j and k such that $A_j \trianglelefteq A_k$. Choose $\mathcal{H}_{i+1}$ such that A_j and A_k are in $\mathcal{H}_{i+1}$. As we have $\mathcal{H}_i \triangleright \mathcal{H}_{i+1}$ we have that $\mathcal{H}_{i+1} \notin \mathcal{W}_{\trianglelefteq}$, a contradiction as we know $A_j \trianglelefteq A_k$. □

2.4.2 Unfold-fold style rewrite rules

The term "unfold-fold" is an umbrella used for a number of different algorithms, all of which rewrite terms into recursive functions by showing that the term is a fixed-point of the function body. This technique is originates from a paper by Burstall and Darlington [16]. Unfold-fold algorithms include deforestation [72, 73], fixed-point fusion [52, 55], supercompilation [68, 40, 66, 6], and Elea's fusion steps, which will be defined in Chapter 5.

For some rewrite relation $\longrightarrow$ all unfold-fold rewrites have more-or-less the shape given in Definition 2.49, which I refer to as the unfold-fold principle.

Definition 2.49 (Unfold-fold principle). For some terms A and H, some variables $x_1...x_n$ free in A, some fresh variable h, and some set of definitions Φ:

$$\frac{\Phi, (h := \text{fn}\ x_1, ..., x_n.\ A) \vdash \text{fn}\ x_1, ..., x_n.\ A \longrightarrow H}{\Phi \vdash A \longrightarrow \text{fix}\,(\text{fn}\ h.\ H)\ x_1...x_n}$$

To explain the above: we choose $x_1...x_n$ from the free variables of A and define a new function variable h to be fn $x_1, ..., x_n.\ A$, a step which manifests as adding $h :=$ fn $x_1, ..., x_n.\ A$ to our available definitions, and means that we are able to rewrite fn $x_1, ..., x_n.\ A \longrightarrow h$ within fn $x_1, ..., x_n.\ A \longrightarrow H$. We then use this newly added rule to rewrite fn $x_1, ..., x_n.\ A$ to H, and from this we are able to rewrite A to fix (fn $h.\ H$) $x_1...x_n$. In supercompilation, rewriting to fn $x_1, ..., x_n.\ A \longrightarrow h$ is referred to as folding, and rewriting fn $x_1, ..., x_n.\ A \longrightarrow H$ is referred to as driving.

This rule shows fn $x_1, ..., x_n.\ A$ to be a fixed-point of fn $h.\ H$, and hence preserves $\sqsupseteq$ (by the least-fixed point principle), a result called the partial correctness theorem [61, 44, 20], but it does not in general preserve $\sqsubseteq$. Since rewriting to a less defined term is rarely desired, every specific form of unfold-fold, such as supercompilation or deforestation, ensures the preservation of $\sqsubseteq$ or equivalence by imposing constraints on the application of the unfold-fold principle. Deforestation, which Section 11.2 discusses, imposes syntactic constraints on the shape of A, to ensure equivalence is preserved.

As a simple example we deforest/supercompile the term $\mathtt{add}\ x\ \mathtt{0}$, where the definition of `add` is given in Appendix A. First we abstract over x and add $\mathtt{add}\ x\ \mathtt{0} \longrightarrow h$ to our list of available rewrites, for some fresh h. This means our folding step does fn $x.\ \mathtt{add}\ x\ \mathtt{0} \longrightarrow h$. Now we "drive" fn $x.\ \mathtt{add}\ x\ \mathtt{0}$,

$$
\begin{array}{l}
\text{fn } x.\ \mathtt{add}\ x\ \mathtt{0} \\
\qquad \longrightarrow \{ \text{ by driving } \} \\
\text{fn } x.\ \text{case } x \text{ of} \\
\quad \mathtt{0} \to \mathtt{0} \\
\quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{add}\ x'\ \mathtt{0}) \\
\qquad \longrightarrow \{ \text{ by folding } \} \\
\text{fn } x.\ \text{case } x \text{ of} \\
\quad \mathtt{0} \to \mathtt{0} \\
\quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (h\ x')
\end{array}
$$

Using the unfold-fold principle with the above gives us:

$$
\mathtt{add}\ x\ \mathtt{0} \longrightarrow \text{fix} \begin{pmatrix} \text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \mathtt{0} \to \mathtt{0} \\ \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (g\ x') \end{pmatrix} x
$$

Chapter 3

Proof by fixed-point promotion

The system I describe in this thesis proves properties of denotational approximation, $A \sqsubseteq B$, using a term rewriting system. These are properties which would traditionally have been shown using a cyclic proof method, such as induction or coinduction. When I say "proof by term rewriting" I mean any proof in which we have been able to rewrite terms to the point that we no longer require a cyclic proof method.

One such technique is employed by the HOSC [40] theorem prover, which proves a term equivalent to another by supercompiling both terms and checking for syntactic equality up to variable renaming. This approach will fail for terms which are equivalent but not syntactically so after supercompilation, such as $\texttt{eq}\,x\,y \cong \texttt{eq}\,y\,x$, where $\texttt{eq}$ is some equality predicate function.

I have developed an alternative approach, which I call proof by fixed-point promotion. Section 3.1 gives a high-level overview of this approach, and Section 3.2 gives an example of such a proof, with a cyclic proof for comparison.

3.1 Overview of fixed-point promotion

This section shows, at a high-level, how my automated prover Elea proves a property of the form: $A \sqsubseteq B$, viz. that A denotationally approximates B. The first part of this technique is to rewrite the left-hand side of $\sqsubseteq$ to have the shape $\text{fix}\,(F)\,x_1 ... x_n$, where $x_1 ... x_n$ are all unique and none are free in F. I refer to this shape as fixed-point promoted form.

My many-step rewrite, which aims to produce fixed-point promoted form, is denoted $\xrightarrow{\sqsubseteq}_+$, so we are trying to get

$$A \xrightarrow{\sqsubseteq}_+ \text{fix}\,(F)\;\; x_1 ... x_n$$

Let's say we perform this rewrite, and that the result is in fixed-point promoted form, viz. $x_1, ..., x_n$ are all unique and none are free in F. I have proven $\xrightarrow{\sqsubseteq}_+$ to preserve $\sqsubseteq$, hence we have

$$A \sqsubseteq \text{fix}\,(F)\;\; x_1 ... x_n$$

From the above and the transitivity of $\sqsubseteq$, to prove $A \sqsubseteq B$ it suffices to prove

$$\mathrm{fix}\,(F)\, x_1 ... x_n \sqsubseteq B$$

Because $x_1 ... x_n$ are all unique and none are free in F, we can rewrite this goal to

$$\mathrm{fix}\,(F) \sqsubseteq (\mathrm{fn}\ x_1, ..., x_n.\ B)$$

The purpose of fixed-point promoted form is that it creates a goal of the above shape - a least fixed-point on the left-hand side of $\sqsubseteq$. This is desirable because of the following least fixed-point principle (Lemma 2.4 on page 32), which states that $\mathrm{fix}\,(F)$ is the smallest X (by $\sqsubseteq$) such that $F\,X \sqsubseteq X$.

$$\forall X\,.\quad F\,X \sqsubseteq X \quad \Rightarrow \quad \mathrm{fix}\,(F) \sqsubseteq X$$

Applying this rule top-down reduces our goal to

$$F\ (\mathrm{fn}\ x_1, ..., x_n.\ B) \sqsubseteq (\mathrm{fn}\ x_1, ..., x_n.\ B)$$

Which can be shown by proving

$$F\ (\mathrm{fn}\ x_1, ..., x_n.\ B)\ x_1 ... x_n \sqsubseteq B$$

Now the process repeats by converting $F\ (\mathrm{fn}\ x_1, ..., x_n.\ B)\ x_1 ... x_n$ into fixed-point promoted form, and continuing until it reaches $B \sqsubseteq B$, which trivially holds. The intention is that every iteration of this process moves the left-hand term closer to being syntactically equal to the right-hand side, since we are substituting the right-hand side into the left when we apply the least fixed-point principle.

Here is a summary of this proof process as a derivation:

$$\dfrac{A \xrightarrow{\sqsubseteq}_{+} \mathrm{fix}\,(F)\ x_1 ... x_n \qquad \dfrac{\dfrac{\overline{B \sqsubseteq B}}{\vdots}}{F\ (\mathrm{fn}\ x_1, ..., x_n.\ B)\ x_1 ... x_n \sqsubseteq B}}{A \sqsubseteq B} \quad \left(\begin{array}{l}\text{if } x_1 ... x_n\\ \text{are all unique}\\ \text{and none free}\\ \text{in } F\end{array}\right)$$

This proof technique is fundamentally a term rewriting technique. The tricky part is the fixed-point promoting rewrite $\xrightarrow{\sqsubseteq}_{+}$, which is why we refer to our overall proof technique as fixed-point promotion. In the next section I will give a concrete example of an approximation proof by fixed-point promotion, along with a proof by induction for comparison.

3.2 Fixed-point promotion vs. cyclic proof

To further explain the fixed-point promotion technique, this section gives an example proof by fixed-point promotion, and then a comparison cyclic proof. Using the definition of `add` from Appendix A, I prove

$$\texttt{add}\ x\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$$

3.2.1 Example proof by fixed-point promotion

The first proof I give of the above property uses fixed-point promotion. Firstly, my rewrite system finds a fixed-point promoted form for the left-hand side:

$$\texttt{add}\ x\ (\texttt{Suc}\ y) \xrightarrow{\sqsubseteq}_{+} \text{fix}\,(\texttt{add-suc}')\ x\ y$$

$$\text{where}\quad \texttt{add-suc}' \overset{\text{def}}{=} \text{fn}\ f, x, y.\ \begin{array}{l}\text{case}\ x\ \text{of}\\ \quad \texttt{0} \to \texttt{Suc}\ y\\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (f\ x'\ y)\end{array}$$

So $\text{fix}\,(\texttt{add-suc}')\ x\ y$ is a fixed-point promoted form of $\texttt{add}\ x\ (\texttt{Suc}\ y)$. We then combine this rewrite with our least pre-fixed-point rule to get our full proof:

$\texttt{add}\ x\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\quad \Leftarrow \{$ by $\xrightarrow{\sqsubseteq}_{+}$ and transitivity of $\sqsubseteq$ $\}$

$\text{fix}\,(\texttt{add-suc}')\ x\ y \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\quad \Leftarrow \{$ by least pre-fixed-point rule (Lemma 2.4 on page 32) $\}$

$\texttt{add-suc}'\ (\text{fn}\ x, y.\ \texttt{Suc}\ (\texttt{add}\ x\ y))\ x\ y \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\quad \Leftrightarrow \{$ by definition of $\texttt{add-suc}'$ and beta-reduction $\}$

$$\left(\begin{array}{l}\text{case}\ x\ \text{of}\\ \quad \texttt{0} \to \texttt{Suc}\ y\\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y))\end{array}\right) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$$

$\quad \Leftrightarrow \{$ by cases on x $\}$

case $x = \bot$:

$\quad \bot \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\quad\quad \Leftrightarrow \{$ by $\forall A\,.\ \bot \sqsubseteq A$ $\}$

$\quad$ true

case $x = \texttt{0}$:

$\quad \texttt{Suc}\ y \sqsubseteq \texttt{Suc}\ (\texttt{add}\ \texttt{0}\ y)$

$\quad\quad \Leftrightarrow \{$ by definition of $\texttt{add}$ and reflexivity of $\sqsubseteq$ $\}$

$\quad$ true

case $x = \texttt{Suc}\ x'$:

$\quad \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y)) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ (\texttt{Suc}\ x')\ y)$

$\quad\quad \Leftrightarrow \{$ by definition of $\texttt{add}$ and reflexivity of $\sqsubseteq$ $\}$

$\quad$ true

3.2.2 Example cyclic proof

Now that a proof of this property using fixed-point promotion has been shown, this section will give a cyclic proof for comparison. This proof could be either induction or coinduction without changing the shape of the proof.

assume $\forall x \,.\, \texttt{add}\ x\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\texttt{add}\ x\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\Leftrightarrow$ { by cases on x }

case $x = \bot$:

$\bot \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$

$\Leftrightarrow$ { by $\forall A \,.\, \bot \sqsubseteq A$ }

true

case $x = \texttt{0}$:

$\texttt{add 0}\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add 0}\ y)$

$\Leftrightarrow$ { by definition of `add` and reflexivity of $\sqsubseteq$ }

true

case $x = \texttt{Suc}\ x'$:

$\texttt{add}\ (\texttt{Suc}\ x')\ (\texttt{Suc}\ y) \sqsubseteq \texttt{Suc}\ (\texttt{add}\ (\texttt{Suc}\ x')\ y)$

$\Leftrightarrow$ { by definition of `add` }

$\texttt{Suc}\ (\texttt{add}\ x'\ (\texttt{Suc}\ y)) \sqsubseteq \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y))$

$\Leftrightarrow$ { by assumption and monotonicity }

$\texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y)) \sqsubseteq \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y))$

$\Leftrightarrow$ { by reflexivity of $\sqsubseteq$ }

true

In terms of automation, there are two differences between the cyclic proof and the fixed-point promotion one. The first can be seen at the case analysis step. The fixed-point promotion based prover does case analysis because it has reached a term with a pattern match top-most in the inequality:

$$\begin{pmatrix} \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x'\ y)) \end{pmatrix} \sqsubseteq \texttt{Suc}\ (\texttt{add}\ x\ y)$$

In comparison, the cyclic prover must analyse the definition of `add` in order to discover on which terms to perform case-analysis, a technique referred to as recursion analysis [38]. A fixed-point promotion based prover does not require this function analysis step.

The second, very important, difference is that the cyclic proof requires the unification of a subterm in a sub-goal, with one side of the assumption, so that it can be applied as a rewrite. The fixed-point promotion proof required no such unification step, since it has no assumption, which is why I refer to fixed-point promotion as an alternative to cyclic proof, rather than a type of cyclic proof. Rewriting a term to the point that an inductive hypothesis (the cyclic assumption in a proof by induction) is applicable as a rewrite, is the difficult part of automating proof by induction, and has been the focus of extensive research [13, 15, 12, 30, 9, 10, 32]. This research is covered in more detail in Section 10.2.4 on page 138.

This unification step with an induction hypothesis has not vanished, but has moved into the fixed-point promoting rewrite at the start of the proof. In essence, fixed-point promotion re-frames the tricky part of cyclic proof, and arguably the tricky part of proving term approximation (or equivalence), as a problem of term rewriting.

As discussed in Chapter 1, the advantage to fixed-point promotion approach, or proof by rewriting in general, is that it simplifies the problem of generalisation. All generalisation steps used in fixed-point promotion occur at the term rewriting stage, and generalisation of a term is a simpler problem than generalisation of a property. Take for example the property used in this chapter, $\texttt{add}\,x\;(\texttt{Suc}\,y) \sqsubseteq \texttt{Suc}\;(\texttt{add}\,x\;y)$ and specialise it by setting y to also be x, giving $\texttt{add}\,x\;(\texttt{Suc}\,x) \sqsubseteq \texttt{Suc}\;(\texttt{add}\,x\;x)$. If a cyclic prover such as Oyster/Clam (Section 10.2.4 on page 138) were to prove this property it would need to generalise these xs back into ys. The difficulty lies in discovering which xs in $\texttt{add}\,x\;(\texttt{Suc}\,x) \sqsubseteq \texttt{Suc}\;(\texttt{add}\,x\;x)$ should be generalised, as an x must be generalised to a y on both sides of the property such that it still holds. For example, $\texttt{add}\,x\;(\texttt{Suc}\,y) \sqsubseteq \texttt{Suc}\;(\texttt{add}\,y\;x)$ would not be a valid generalisation. In comparison, the Elea prover has to generalise only the term $\texttt{add}\,x\;(\texttt{Suc}\,x)$, for which it can replace either occurrence of x with a fresh variable.

Chapter 4

Preliminaries of fixed-point promotion

The previous chapter shows how rewriting a term into fixed-point promoted form allows us to prove properties of denotational approximation by using the least fixed-point principle, as opposed to a cyclic proof rule such as induction. The main result of this research is a term rewriting system to produce fixed-point promoted form, and this chapter describes the basics of this system, including its simplest rewrite rules, such as beta reduction and fixed-point unrolling. The rewrite steps which actually produce fixed-point promoted form are the truncation fusion rules given in the next chapter.

I describe this rewriting system as a numbered list of rewrite rules. Each Rule defines a judgement of the form:

$$\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B$$

The above states that term A can be rewritten in one step to term B, given rewrite environment $(\Gamma, \Phi, \mathcal{H})$. The relation variable $\mathfrak{R}$ can be either $\sqsubseteq$ or $\sqsupseteq$ and tells us what ordering this rewrite step is proven to preserve. So (ignoring environment), if we have a rule $A \xrightarrow{\sqsubseteq} B$, we know that $A \sqsubseteq B$. Similarly, a rewrite of the shape $A \xrightarrow{\sqsupseteq} B$ means we know $B \sqsubseteq A$. If a rule is defined in terms of our relation variable $A \xrightarrow{\mathfrak{R}} B$, then we have proven both $A \sqsubseteq B$ and $B \sqsubseteq A$, since $\mathfrak{R}$ can be either $\sqsubseteq$ or $\sqsupseteq$.

The reason Elea needs two different rewrite rules, one for $\sqsubseteq$ and one for $\sqsupseteq$, is that it is designed to prove $\sqsubseteq$ properties. In proving $A \sqsubseteq B$, $\xrightarrow{\sqsubseteq}$ allows us to rewrite the left-hand term and $\xrightarrow{\sqsupseteq}$ allows us to rewrite the right-hand one. To elaborate, $A \xrightarrow{\sqsubseteq} A'$, and hence $A \sqsubseteq A'$, means $A' \sqsubseteq B \Rightarrow A \sqsubseteq B$. Since we are developing a top-down theorem prover $A \xrightarrow{\sqsubseteq} A'$ allows us to reduce the property $A \sqsubseteq B$ to the sufficient property $A' \sqsubseteq B$. Similarly, $B \xrightarrow{\sqsupseteq} B'$, and hence $B' \sqsubseteq B$, means $A \sqsubseteq B' \Rightarrow A \sqsubseteq B$ and so we can reduce proving $A \sqsubseteq B$ to proving $A \sqsubseteq B'$.

The rest of this chapter is broken down as follows:

(Section 4.1) Describes the three variables of the rewrite environment: type environment Γ, fact environment Φ, and history environment $\mathcal{H}$. It also describes formally what I mean when I say this rewriting system is sound, since it is related to Φ.

(Section 4.2)	Gives the rules used to lift the one-step rewrite rules $\xrightarrow{\mathfrak{R}}$ into a terminating, many-step, rewrite rule $\xrightarrow{\mathfrak{R}}_{+}$. It uses the well-founded transitive closure technique from page 40 to ensure termination.
(Section 4.3)	Explains how I have turned this non-deterministic rewriting system into a deterministic algorithm for our Elea tool. This section can be summarised as, given a non-deterministic choice of rewrite rules, pick the one with the lowest number. For this reason each rewrite rule has a number, representing its priority in the deterministic algorithm.
(Section 4.4)	Once the preliminaries of this rewrite system have been covered the rest of this chapter defines its first rewrite rules. While the overall goal is fixed-point promoted form, the rewrites given here do not directly produce terms of this shape. These are the more run-of-the-mill rewrites, such as beta reduction and fixed-point unrolling. It is the fusion rules in the next chapter which produce fixed-point promoted form, but these couldn't function without the rewrites given in this section.

4.1 Environment variables $\Gamma, \Phi, \mathcal{H}$

As with many term rewriting systems, this one carries around environment variables, which are extended as we move further inside terms. $\xrightarrow{\mathfrak{R}}$ uses three separate environment variables, Γ for types, Φ for facts, and $\mathcal{H}$ for previously seen terms.

The type environment Γ has already been used in presenting νPCF; it is a mapping from νPCF variables to νPCF types, which stores the type of any free variables. The history environment $\mathcal{H}$ has also already been presented in Section 2.4 on page 39; it is a sequence of terms to store those we have already encountered in the rewriting process, so that rewriting will terminate when it reaches a term it has already "seen", viz. when $\mathcal{H} \in \mathcal{W}_{\trianglelefteq}$.

The new environment variable introduced here is the fact environment Φ. It stores a list of $A \sqsubseteq B$ properties between terms which are known to hold in this context. Definition 4.2 gives the denotational semantics of Φ as a predicate on value environments.

Definition 4.1 (Fact environment Φ).

$$\Phi \ ::= \ \emptyset \ | \ \Phi,\ A \sqsubseteq B$$

The fact environment Φ is a relation between terms, defined with the above grammar. $(A \sqsubseteq B) \in \Phi$ means we know $A \sqsubseteq B$ within this environment.

Definition 4.2 (Denoting our fact environment $[\![\Phi]\!]$).

$$\begin{aligned} [\![\emptyset]\!]\,\rho \quad & \overset{\text{def}}{\Leftrightarrow} \quad \text{true} \\ [\![\Phi, A \sqsubseteq B]\!]\,\rho \quad & \overset{\text{def}}{\Leftrightarrow} \quad [\![A]\!]\,\rho \sqsubseteq [\![B]\!]\,\rho \ \wedge \ [\![\Phi]\!]\,\rho \end{aligned}$$

Given a type environment Γ which well-types all terms within Φ, $[\![\Phi]\!]$ denotes a predicate over value environments, which holds if all the properties $A \sqsubseteq B \in \Phi$ hold in that value environment.

4.1.1 Soundness

Here is briefly described what I prove in Chapter 9 when I show this system to be sound. Recall from Section 2.3.2 that, if $\Gamma \vdash A : \tau$, then the denotation of the term $[\![A]\!]$ is a function from an assignment of the free variables of A, given by a value environment $\rho \in [\![\Gamma]\!]$, to the domain given by its type, $[\![\tau]\!]$. Definition 4.2 above gives the denotation of our fact environment, $[\![\Phi]\!]$, as a predicate over a Γ-environment, which holds if all the $\sqsubseteq$ properties within it hold for that assignment of free variables.

Using these concepts, I say that a rewrite rule is sound if, for $\mathfrak{R} = \sqsubseteq$ or $\mathfrak{R} = \sqsupseteq$:

$$\begin{array}{ll} \text{if} & \Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \\ \text{then} & \forall (\rho \in [\![\Gamma]\!]) \, . \, [\![\Phi]\!]\, \rho \;\Rightarrow\; [\![A]\!]\, \rho \, \mathfrak{R} \, [\![B]\!]\, \rho \end{array}$$

This is proven formally in Section 9.2 on page 125.

4.2 Many-step rewrite $\xrightarrow{\mathfrak{R}}_+$

The rest of this thesis adds many rules to the one-step rewrite relation $\xrightarrow{\mathfrak{R}}$. This section gives the only two rules which define the many-step rewrite relation $\xrightarrow{\mathfrak{R}}_+$. This definition uses the well-founded transitive closure technique (see page 40) to ensure termination. The well-quasi-order used as a whistle is the homeomorphic embedding, $\trianglelefteq$, given in Definition 2.47 on page 41.

Definition 4.3 ($\xrightarrow{\mathfrak{R}}_+$).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B}{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_+ B}$$

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \qquad \Gamma, \Phi, (\mathcal{H}, A) \vdash B \xrightarrow{\mathfrak{R}}_+ C}{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_+ C} \quad \text{if } \; (\mathcal{H}, A) \notin \mathcal{W}_{\trianglelefteq}$$

These two rules inductively define the $\xrightarrow{\mathfrak{R}}_+$ judgement. It is the same technique as Definition 2.46 on page 40, extended to the environment $\Gamma, \Phi, \mathcal{H}$.

Since $\mathfrak{R}$ is a relation variable, the above definition gives us two judgements, one for $\mathfrak{R} = \sqsubseteq$ and one for $\mathfrak{R} = \sqsupseteq$. As with the one-step rewrites, I have proven (ignoring environment) that if $A \xrightarrow{\sqsubseteq}_+ B$, then $A \sqsubseteq B$, and similarly, if we can apply $A \xrightarrow{\sqsupseteq}_+ B$, we have $B \sqsubseteq A$.

4.3 Turning $\xrightarrow{\mathfrak{R}}_+$ into a deterministic algorithm

In order to use the rewrite relation $\xrightarrow{\mathfrak{R}}_+$ as the algorithm for my Elea tool, it needs to be turned into a left-total functional relation, viz. we must restrict it so that for every term L there exists exactly one term R that L rewrites to.

Definition 4.4 ($\xrightarrow{\mathfrak{R}}_!$).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_! A} \quad \text{if } \; \mathcal{H} \in \mathcal{W}_{\trianglelefteq} \;\vee\; \nexists B \,.\, \Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\nu} B$$

first applicable rewrite rule

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \qquad \Gamma, \Phi, (\mathcal{H}, A) \vdash B \xrightarrow{\mathfrak{R}}_! C}{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_! C} \quad \text{if } \; \mathcal{H} \notin \mathcal{W}_{\trianglelefteq}$$

The left-total functional relation $\xrightarrow{\mathfrak{R}}_!$ is the rewriting algorithm used by Elea and is inductively generated by these two rules. All rewrite rules have an associated number and this is used to prioritise rewrites in order to deterministically choose which rule to use next.

The rest of the thesis uses the rewrite rule $\xrightarrow{\mathfrak{R}}_+$ instead of $\xrightarrow{\mathfrak{R}}_!$ so that I can represent partial rewriting in examples. If you want to extract the deterministic algorithm simply replace every instance of $\xrightarrow{\mathfrak{R}}_+$ with $\xrightarrow{\mathfrak{R}}_!$ in the presentation of these rules.

4.4 Preliminary rewrite rules

Now that the preliminary concepts involved in Elea's rewriting algorithm have been described, much of the rest of this thesis will be concerned with defining the different cases of the one step rewrite $\xrightarrow{\mathfrak{R}}$, from which this algorithm is built. The rules described in this section are the simple ones. They do not directly produce fixed-point promoted form, which is the overall goal of this rewriting system, but they aid the rules in the next chapter in doing so. They are given as follows.

(Section 4.4.1)	Reduction rewrites: beta, eta, and case.
(Section 4.4.2)	Rewrites which move pattern matches into more sensible positions.
(Section 4.4.3)	Three rules which are able to remove pattern matches entirely.
(Section 4.4.4)	A rule which replaces undefined terms with the explicitly undefined term $\bot$.
(Section 4.4.5)	Gives rewrite rules which apply $\xrightarrow{\mathfrak{R}}_+$ to sub-terms.
(Section 4.4.6)	A rewrite rule which uses facts from Φ to reduce a pattern match.
(Section 4.4.7)	Two rewrite rules which unfold least fixed-points and truncated fixed-points respectively.

4.4.1 Reduction rewrites

Rule 1 (beta reduction).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash (\text{fn } x : \tau .\, F)\; A \xrightarrow{\mathfrak{R}} F[A/x]}$$

Rule 2 (eta reduction).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \text{fn } x : \tau.\, F\ x \xrightarrow{\mathfrak{R}} F}$$

$$\text{if } x \notin \mathsf{freeVars}\,(F)$$

Rule 3 (case reduction).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \begin{pmatrix} \text{case con}_i\langle \boldsymbol{T}\rangle\ A_1...A_n \text{ of} \\ P_1 \to B_1 \ldots P_m \to B_m \end{pmatrix} \xrightarrow{\mathfrak{R}} B_j[A_1/x_1]...[A_n/x_n]}$$

$$\text{if} \quad j = \mathsf{findPattern}_i\,(P_1, ..., P_m)$$
$$K\ x_1...x_n = P_j$$

The meta-level function findPattern is given in Definition 2.11 on page 22. The call $\mathsf{findPattern}_i\,(P_1, ..., P_m)$ will return the index of the first occurrence of the ith constructor of $\boldsymbol{T}$ in the sequence $P_1...P_m$, viz. the first pattern which matches $\text{con}_i\langle \boldsymbol{T}\rangle$.

4.4.2 Floating pattern matches

The rewrites here pull pattern matches out of the awkward places they can get themselves into and float them topmost in the term. First though, I need to specify the operator strictArgs, as it is used by one of the rules in this section, then I can go on to describe these rewrites.

A function term F is strict in its nth argument if, given an undefined value ($\bot$) for that argument, the result of the function is undefined, regardless of its other inputs. The meta-level function strictArgs is given in Definition 4.5.

Definition 4.5 (Finding strict arguments).

$$\mathsf{strictArgs}\, F = \{\ i \mid F\ x_1...x_{i-1}\ \bot\ x_i...x_{n-1} \xrightarrow{\sqsubseteq}_! \bot \text{ where } x_1...x_{n-1} \text{ fresh } \}$$

The strictArgs function attempts to give the indexes of all strict arguments to the provided νPCF function. The $\xrightarrow{\sqsubseteq}_!$ rewrite used is a restricted form of the the Elea rewriting system, which only uses the rewrites given in this section. This rewrite is also missing its environment $\Gamma, \Phi, \mathcal{H}$, which will be inherited from the context from which strictArgs is called.

The important property of strictArgs is that it never returns a false positive. While it will not always enumerate every strict argument to a function (which is undecidable in general), if $i \in \mathsf{strictArgs}\, F$ then $[\![F]\!]\,\rho$ is strict in its ith argument. This property follows trivially from the soundness of Elea's rewrite rules.

Rule 4 (float case-case).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \left(\begin{array}{l}\text{case} \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to A_1 \ldots \\ \quad P_n \to A_n\end{array}\right) \text{ of} \\ \quad p'_1 \to B_1 \ldots \\ \quad p'_m \to B_m\end{array}\right) \xrightarrow{\mathfrak{R}} \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to \left(\begin{array}{l}\text{case } A_1 \text{ of} \\ \quad p'_1 \to B_1 \ldots \\ \quad p'_m \to B_m\end{array}\right) \ldots \\ \quad P_n \to \left(\begin{array}{l}\text{case } A_n \text{ of} \\ \quad p'_1 \to B_1 \ldots \\ \quad p'_m \to B_m\end{array}\right)\end{array}\right)}$$

A pattern match over a pattern match can be rewritten to have the inner match topmost.

Rule 5 (float case-fun).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to F_1 \ldots \\ \quad P_n \to F_n\end{array}\right) A \xrightarrow{\mathfrak{R}} \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to F_1\ A \ldots \\ \quad P_n \to F_n\ A\end{array}\right)}$$

This rule takes a pattern match applied as a function and floats it to be topmost.

Rule 6 (float case-arg).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash F\ A_1 \ldots A_{i-1} \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to B_1 \ldots \\ \quad P_n \to B_n\end{array}\right) \xrightarrow{\mathfrak{R}} \left(\begin{array}{l}\text{case } M \text{ of} \\ \quad P_1 \to F\ A_1 \ldots A_{i-1}\ B_1 \ldots \\ \quad P_n \to F\ A_1 \ldots A_{i-1}\ B_n\end{array}\right)}$$

$$\text{if} \quad i \in \mathsf{strictArgs}\ F$$

This rule takes a pattern match applied as an argument and floats it topmost. It requires the strictness condition, as otherwise it could produce a term less defined than the original.

4.4.3 Removing pattern matches

This section gives a couple of rewrites which are able to remove pattern matches altogether.

Rule 7 (identity case).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \left(\text{case } M \text{ of } P_1 \to P_1 \ldots P_n \to P_n\right) \xrightarrow{\mathfrak{R}} M}$$

Removes a pattern match which just returns the term it is matching on.

Rule 8 (constant case).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \big(\text{case } M \text{ of } P_1 \to A \,...\, P_n \to A\big) \xrightarrow{\sqsubseteq} A}$$

This rule removes a pattern match if every branch returns the same value. This may not preserve denotational equivalence, as A may be more defined than M; hence it is only defined for $\mathfrak{R} = \sqsubseteq$.

4.4.4 Rewriting to $\bot$

This section adds Rule 9 (undefined), which performs a simple check to see if a term is denotationally equivalent to undefinedness and, if so, rewrites it to $\bot$. The $\mathsf{is}_\bot\,(...)$ operator from Definition 4.6 is the lightweight undefinedness check.

Definition 4.6 ($\mathsf{is}_\bot\,(...)$).

$$\begin{aligned}
\mathsf{is}_\bot\,(\bot) &\overset{\text{def}}{=} \text{true} \\
\mathsf{is}_\bot\,(\bot\; A_1 ... A_n) &\overset{\text{def}}{=} \text{true} \\
\mathsf{is}_\bot\,(\text{fn } x.\, \bot) &\overset{\text{def}}{=} \text{true} \\
\mathsf{is}_\bot\,(F\; A_1 ... A_n) &\overset{\text{def}}{=} \textstyle\bigvee_{i \in \mathsf{strictArgs}(F)} \mathsf{is}_\bot\,(A_i) \\
\mathsf{is}_\bot \begin{pmatrix} \text{case } M \text{ of} \\ \quad P_1 \to A_1 \,... \\ \quad P_n \to A_n \end{pmatrix} &\overset{\text{def}}{=} \mathsf{is}_\bot\,(M) \vee \textstyle\bigwedge_{i \in [1..n]} \mathsf{is}_\bot\,(A_i) \\
\mathsf{is}_\bot\,(A) &\overset{\text{def}}{=} \text{false}
\end{aligned}$$

$\mathsf{is}_\bot\,(E)$ is a lightweight check that E is denotationally equivalent to $\bot$, specified in Definition 4.6. The operator $\mathsf{strictArgs}$ (Definition 4.5) has been proven to return a subset of the indices of the strict arguments for the given function. If an argument at one of these indices is undefined, then the entire term will be undefined.

Rule 9 (undefined).

$$\frac{\Gamma \vdash A : \tau}{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} \bot_\tau} \quad \text{if} \quad \mathsf{is}_\bot\,(A)$$

This rule detects terms denotationally equivalent to $\bot$ and rewrites them as such. The term $\bot_\tau$ is given in Definition 2.14 on page 23.

4.4.5 Rewriting sub-terms

The rules in the previous section were simple ones which preserve denotational equality, and hence were defined for $\xrightarrow{\mathfrak{R}}$, where $\mathfrak{R}$ can be $\sqsubseteq$ or $\sqsupseteq$. The rules in this section recursively apply $\xrightarrow{\mathfrak{R}}$ to the sub-terms of its target, and so will preserve whichever ordering the sub-term rewrite preserves.

These rules are defined in terms of $\xrightarrow{\mathfrak{R}}_{+}$ instead of $\xrightarrow{\mathfrak{R}}$ in order to accurately replicate the behaviour of Elea, which will rewrite sub-terms as much as possible when applying traverse rules. These rules preserve termination, even without adding a term to $\mathcal{H}$, as the antecedent rewrite is performed on a sub-term and our termination ordering allows $\mathcal{H}$ to stay the same, as long as term size shrinks.

Rule 10 (traverse match).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash M \xrightarrow{\mathfrak{R}}_{+} M'}{\Gamma, \Phi, \mathcal{H} \vdash (\text{case } M \text{ of } ...) \xrightarrow{\mathfrak{R}} (\text{case } M' \text{ of } ...)}$$

Rule 11 (traverse lambda).

$$\frac{\Gamma[x \mapsto \tau], \Phi, \mathcal{H} \vdash F \xrightarrow{\mathfrak{R}}_{+} F'}{\Gamma, \Phi, \mathcal{H} \vdash \text{fn } x : \tau.\, F \xrightarrow{\mathfrak{R}} \text{fn } x : \tau.\, F'}$$

Rule 12 (traverse argument).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_{+} A'}{\Gamma, \Phi, \mathcal{H} \vdash F\ A \xrightarrow{\mathfrak{R}} F\ A'}$$

Rule 13 (traverse function).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash F \xrightarrow{\mathfrak{R}}_{+} F'}{\Gamma, \Phi, \mathcal{H} \vdash F\ A \xrightarrow{\mathfrak{R}} F'\ A}$$

Rule 14 (traverse fix).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash F \xrightarrow{\mathfrak{R}}_{+} F'}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(F) \xrightarrow{\mathfrak{R}} \text{fix}\,(F')}$$

Rule 15 (traverse tagged fix).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash F \xrightarrow{\mathfrak{R}}_{+} F'}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^{a}\,(F) \xrightarrow{\mathfrak{R}} \text{fix}^{a}\,(F')}$$

Rule 16 (traverse var-branch).

$$\frac{(\Gamma \lhd\ P_i), \Phi[P_i/x], \mathcal{H} \vdash A_i \xrightarrow{\mathfrak{R}}_{+} A_i'}{\Gamma, \Phi, \mathcal{H} \vdash (\text{case } x \text{ of } ...\ P_i \to A_i\ ...) \xrightarrow{\mathfrak{R}} (\text{case } x \text{ of } ...\ P_i \to A_i'\ ...)}$$

The $(\Gamma \lhd\ p)$ operator is given on page 21, and adds all the variables bound in the pattern p to the type environment Γ. In the case of an else pattern, we treat $[\text{else}/x]$ as the identity substitution. Applying the substitution $[P_i/x]$ to our stored rewrite rules applies the substitution to the term on either side of every rewrite rule in the list:

$$(\Phi,\ A \sqsubseteq B)\sigma \quad \overset{\text{def}}{=} \quad \Phi\sigma,\ A\sigma \sqsubseteq B\sigma$$

Rule 17 (traverse branch).

$$\frac{(\Gamma \lhd\ P_i), (\Phi, M \sqsubseteq P_i, P_i \sqsubseteq M), \mathcal{H} \vdash A_i \xrightarrow{\mathfrak{R}}_{+} A_i'}{\Gamma, \Phi, \mathcal{H} \vdash (\text{case } M \text{ of } ...\ P_i \to A_i\ ...) \xrightarrow{\mathfrak{R}} (\text{case } M \text{ of } ...\ P_i \to A_i'\ ...)}$$

This rule is the same as the above, except that we are matching on a non-variable term; so, instead of substituting we add this match as a rewrite in both directions.

4.4.6 Using Φ to reduce pattern matches

The rule here can reduce a pattern match using a fact stored within Φ. This fact will have been added by the traverse branch rule from the previous section.

Rule 18 (apply pattern).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \begin{pmatrix} \text{case } M \text{ of} \\ \quad P_1 \to A_1\ ...\ P_n \to A_n \end{pmatrix} \xrightarrow{\mathfrak{R}} A_j[x_1/y_1]...[x_m/y_m]}$$

$$\begin{aligned} \text{if} \quad & (M \sqsubseteq \text{con}_i\langle \boldsymbol{T} \rangle\ x_1...x_m) \in \Phi \\ & j = \mathsf{findPattern}_i\,(P_1, ..., P_n) \\ & K\ y_1...y_m = P_j \end{aligned}$$

Here we are pattern matching on M, but we have in Φ that M can be rewritten to the term $K\ x_1...x_n$, so we can use this fact to reduce the pattern match.

4.4.7 Unfolding fixed-points

The rules given here unfold fixed-points. They are defined in terms of a many-step antecedent rewrite $\xrightarrow{\mathfrak{R}}_+$ because of the not-embedded check each one has, the $A \not\trianglelefteq B$ condition, where $\trianglelefteq$ is given in Definition 2.47 on page 41. This check would always fail if it was made immediately after the unfolding, since $\text{fix}\,(F) \trianglelefteq F\ (\text{fix}\,(F))$, but might not fail after the unfolded term has been rewritten further.

The reason I include this check is that unfolding is always applicable, but not always necessary; without it, every application of unfolding to the same term will cause it to grow again. Though this algorithm would still terminate, we end up with gigantic terms filled with unnecessary unfoldings, which slows down our tool and in some cases blocks other rewrites from being applicable.

This technique controls the potential of such a “code explosion”, as it stops these unfoldings from blocking necessary rewrites in our test cases. It does not completely control the problem, and Elea will still occasionally generate huge function definitions in the course of its rewrites.

There are heuristics used in program optimising supercompilation which can deal with this issue [36], but I did not consider adding them a priority, since these huge function definitions only slow down Elea, but do not stop it completing a proof. Program optimisation suffers far more from code size explosion than does theorem proving, since it leads to over-large binaries.

Rule 19 (unfold fix).

$$\frac{\Gamma, \Phi, (\mathcal{H}, \text{fix}\,(F)) \vdash F\ (\text{fix}\,(F))\ A_1 ... A_n \xrightarrow{\mathfrak{R}}_+ B}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(F)\ A_1 ... A_n \xrightarrow{\mathfrak{R}} B}$$

$$\text{if} \quad \text{fix}\,(F) \not\trianglelefteq B$$

This rule must add $\text{fix}\,(F)$ to $\mathcal{H}$ to ensure termination, as our term size has increased in the antecedent rewrite.

Rule 20 (unfold truncated-fix).

$$\frac{\Gamma, \Phi, (\mathcal{H}, \text{fix}^a\,(F)) \vdash F\ (\text{fix}^a\,(F))\ A_1 ... A_n \xrightarrow{\sqsubseteq}_+ B}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^a\,(F)\ A_1 ... A_n \xrightarrow{\sqsubseteq} B}$$

$$\text{if} \quad \text{fix}^a\,(F) \not\trianglelefteq B$$

Recall that $\text{fix}^a\,(F)$ is a fixed-point which can only be unrolled a many times, where a is a natural number meta-variable. Due to this, if we rewrite $\text{fix}^a\,(F)$ to $F\ (\text{fix}^a\,(F))$, we are producing a more defined term than the original, as it has been unrolled without decreasing the a value. Hence, this rewrite is only defined for $\mathfrak{R} = \sqsubseteq$, unlike unfold fix which preserves denotational equivalence.

There are additional heuristics within Elea which control the applicability of the above two rules. Specifically, these rules are only applicable in one of the following situations:

1. All strict arguments to the fixed-point have constructors topmost.
2. Any strict arguments to the fixed-point contain only constructors, viz. no variables or fixed-points.
3. This term is within a pattern match and the fixed-point produces a constructor down every return branch, viz. it is a productive fixed-point.
4. This term is pattern matched within an assertion, and one argument contains a constructor topmost.

Chapter 5

Fusion

The overall aim of this thesis is an automated proof system for properties of denotational approximation between terms in νPCF. Section 3.1 demonstrated how we can prove such properties by rewriting a term into fixed-point promoted form (FPPF) so that we can then apply the least fixed-point principle. The strength of this proof method comes from its rewrite system, which produces terms in FPPF. The preliminary rewrites of this system were given in Chapter 4, but none of these rules directly produced such terms.

It is the rewrite rules described in this chapter which produce fixed-point promoted form, all of which are based on the principle of fusion. This chapter is broken down as follows:

(Section 5.1)	A high-level overview of what fusion is, and how it rewrites terms into FPPF.
(Section 5.2)	This fusion process happens in two stages: detecting when fusion would be applicable and then applying fusion. Here we give the two rewrite rules which apply fusion: ω-fusion (for $\xrightarrow{\sqsupseteq}$) and truncation fusion (for $\xrightarrow{\sqsubseteq}$).
(Section 5.2.1)	The above fusion rule adds a fact to our fact environment, in this section we describe the rewrite rule which applies this fact: folding, named as such as it matches the folding step of a supercompiler. The rest of this chapter describes the rules which detect when the truncation fusion rule from the previous section can be applied.
(Section 5.3)	The constant-argument fusion rule fuses an argument into the body of a fixed-point. It is this rule which allows Elea to rewrite $\texttt{app}\; xs\; \texttt{[}y\texttt{]} \xrightarrow{\mathfrak{R}}_+ \texttt{snoc}_y\; xs$ (definitions are given in Appendix A).
(Section 5.4)	The fix-fix fusion rule fuses a fixed-point into its own argument, if that argument is itself a fixed-point. It is this rule which allows Elea to rewrite $\texttt{rev}\; (\texttt{rev}\; xs) \xrightarrow{\sqsubseteq}_+ xs$.
(Section 5.5)	Most of the time, having a constructor term as an argument to a fixed-point indicates that this fixed-point should be unfolded. Sometimes, unfolding is not applicable, as the not-embedded check will fail (see Rule 19 on page 60). An example of this is the term $\texttt{eq}\; x\; (\texttt{Suc}\; x)$. For cases like this, Elea has the constructor fusion rule, which allows it to perform the following rewrite $\texttt{eq}\; x\; (\texttt{Suc}\; x) \xrightarrow{\sqsubseteq}_+ \texttt{False}$.

(Section 5.6) One of the requirements of fixed-point promoted form is that all of the variable arguments to the fixed-point are unique. This section describes the repeated-variable fusion rule which can remove occurrences of non-unique arguments to a fixed-point. It is this rule which allows Elea to rewrite $\texttt{eq}\,x\,x \xrightarrow{\sqsubseteq}_{+} \texttt{True}$.

(Section 5.7) Another requirement of FPPF is that no arguments to a fixed-point are free variables within the fixed-point. If we encounter a free variable within a fixed-point, which also occurs as an argument to that fixed-point, we can use the free-variable fusion rule described here to capture the free variable as the argument variable. It is this rule which allows Elea to rewrite $\texttt{add}_x\,x \xrightarrow{\sqsubseteq}_{+} \texttt{double}\,x$.

(Section 5.8) Some fixed-point arguments neither decrease nor stay constant in recursive calls. These are known as accumulating arguments and are very difficult for cyclic provers to reason about. This section describes accumulation fusion, which attempts to fuse a non-variable accumulating argument into a fixed-point. It is this rule which gives Elea the rewrite $\texttt{it-rev}\,xs\;\texttt{[ ]} \xrightarrow{\sqsubseteq}_{+} \texttt{rev}\,xs$.

(Section 5.9) fact fusion replaces unreachable branches in fixed-points with $\bot$ where this unreachability is inferred from the facts within Φ. Unlike our previous fusion steps, its aim is not fixed-point promoted form, but to replace enough branches that the entire fixed-point can be collapsed to a constant using subterm fission. It is this rule which allows Elea to rewrite $\texttt{or}\;(\texttt{le}\,x\,y)\;(\texttt{le}\,y\,x) \xrightarrow{\sqsubseteq}_{+} \texttt{or}\;(\texttt{le}\,x\,y)\;\texttt{True}$.

5.1 How fusion produces fixed-point promoted form

A term is in fixed-point promoted form if it has the shape $\mathrm{fix}\,(F)\,x_1...x_n$, where all $x_1...x_n$ are unique and none are free in F. Every rule which applies fusion aims to remove an anti-pattern to fixed-point promoted form. An example of such an anti-pattern would be a variable argument which is also free in the fixed-point (removed by free-variable fusion), or if a fixed-point call is an argument to another fixed-point (removed by fix-fix fusion).

Each rule isolates such an anti-pattern by representing it as a term context $\mathcal{C}$ and fusing it into a fixed-point $\mathrm{fix}\,(G)$, resulting in a new fixed-point $\mathrm{fix}\,(H)$, such that $\mathcal{C}[\mathrm{fix}\,(G)]\;\mathfrak{R}\;\mathrm{fix}\,(H)$, where $\mathfrak{R} = \sqsubseteq$ or $\sqsupseteq$. I represent this fusion process with a new shape of rewrite $\mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\mathfrak{R}} \mathrm{fix}\,(H)$.

For example, let's say we have the following term

$\texttt{eq}\,x\,x$ (eq is given in Definition 5.1)

There is one anti-pattern here, the repeated x argument to our equality function. The rewrite repeated-variable fusion removes this repeated argument by expressing it as the context $\mathrm{fn}\;x.\,\Box\,x\,x$ and fusing this into the fixed-point eq.

$$\frac{(\mathrm{fn}\;x.\,\Box\,x\,x) \oplus \texttt{eq} \xrightarrow{\mathfrak{R}} \texttt{eq-refl}}{\texttt{eq}\,x\,x \xrightarrow{\mathfrak{R}} \texttt{eq-refl}\,x} \qquad (\texttt{eq-refl}\text{ given in Definition 5.1 on page 65})$$

An example which requires multiple fusion rewrites is $\texttt{len}\,(\texttt{app}\,ys\,ys)$, where `len` is a list length function and `app` is list append, both given in Definition 5.1. The first anti-pattern we isolate is the second ys argument. This second argument to the `app` function is something I refer to as a constant argument, and its value can be pushed inside the body of the fixed-point itself, with the following constant-argument fusion rewrite.

$$\frac{(\text{fn } xs.\, \square\, xs\; ys) \oplus \texttt{app} \xrightarrow{\mathfrak{R}} \texttt{app}_{ys}}{\texttt{len}\,(\texttt{app}\; ys\; ys) \xrightarrow{\mathfrak{R}} \texttt{len}(\texttt{app}_{ys}\; ys)}$$

The next anti-pattern we can remove is the `len` fixed-point surrounding the call to $\texttt{app}_{ys}\; ys$. Expressing this as the context fn $xs.\, \texttt{len}\,(\square\; xs)$ allows us to fuse this outer fixed-point into the $\texttt{app}_{ys}$ fixed-point. Notice that we have generalised the ys argument to some new xs, so that the value of ys in the body of $\texttt{app}_{ys}$ does not clash with the fusion process. The context we are fusing in expresses only one anti-pattern, the outer fixed-point call, generalising all others. This shape of rewrite I refer to as fix-fix fusion.

$$\frac{(\text{fn } xs.\, \texttt{len}\,(\square\, xs)) \oplus \texttt{app}_{ys} \xrightarrow{\mathfrak{R}} \texttt{len-app}_{ys}}{\texttt{len}\,(\texttt{app}_{ys}\; ys) \xrightarrow{\mathfrak{R}} \texttt{len-app}_{ys}\; ys}$$

We are now down to our final anti-pattern! Recall that in fixed-point promoted form all variable arguments must not be free within the body of the fixed-point, so we must capture the ys variable that is free within $\texttt{len-app}_{ys}$. This can be done by fusing the context fn $ys.\, \square\, ys$ into $\texttt{len-app}_{ys}$, since fn ys will capture ys. I refer to a rewrite of this shape as free-argument fusion.

$$\frac{(\text{fn } ys.\, \square\, ys) \oplus \texttt{len-app}_{ys} \xrightarrow{\sqsubseteq} \texttt{double-len}}{\texttt{len-app}_{ys}\; ys \xrightarrow{\sqsubseteq} \texttt{double-len}\; ys}$$

We have to switch to $\sqsubseteq$ in the above rewrite, as this fusion step internally requires a rewrite which preserves $\sqsubseteq$, but not $\sqsupseteq$ (constructor fission, which will be given in Section 6.3 on page 91).

Definition 5.1 (Term macros used above).

$$\texttt{eq} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, x, y.\ \text{case } x, y \text{ of} \\ \quad \texttt{0}, \texttt{0} \to \texttt{True} \\ \quad \texttt{0}, \texttt{Suc}\, y' \to \texttt{True} \\ \quad \texttt{Suc}\, x', \texttt{0} \to \texttt{False} \\ \quad \texttt{Suc}\, x' \to f\ x'\end{pmatrix}$$

$$\texttt{eq-refl} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{True} \\ \quad \texttt{Suc}\, x' \to f\ x'\end{pmatrix}$$

$$\texttt{app} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, xs, ys.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to x :: f\ xs'\ ys\end{pmatrix}$$

$$\texttt{app}_{ys} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to x :: f\ xs'\end{pmatrix}$$

$$\texttt{len-app}_{ys} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{len}\, ys \\ \quad x :: xs' \to \texttt{Suc}\,(f\ xs')\end{pmatrix}$$

$$\texttt{double-len} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{0} \\ \quad x :: xs' \to \texttt{Suc}\,(\texttt{Suc}\,(f\ xs'))\end{pmatrix}$$

The fusion process can be summarised as, on encountering a term containing a fixed-point, but not in FPPF, $\mathcal{C}[\text{fix}\,(G)]$ for example, we first isolate each different FPPF anti-pattern in the context $\mathcal{C}$ as a family of contexts $\mathcal{C}_1 ... \mathcal{C}_n$, with a set of free variables $x_1 ... x_m$, such that

$$\mathcal{C}[\text{fix}\,(G)] \;\cong\; \mathcal{C}_n[...\,\mathcal{C}_2[\mathcal{C}_1[\text{fix}\,(G)]]\,...]\ x_1 ... x_m$$

We then successively apply fusion to every anti-pattern context $\mathcal{C}_i$ until they are all fused into the fixed-point $\text{fix}\,(G)$, leaving us with a term in fixed-point promoted form.

$$\frac{\mathcal{C}_1 \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(G_2) \;\; \mathcal{C}_2 \oplus \text{fix}\,(G_2) \xrightarrow{\mathfrak{R}} \text{fix}\,(G_3) \;\; ...\; \mathcal{C}_n \oplus \text{fix}\,(G_n) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\mathcal{C}_n[...\,\mathcal{C}_2[\mathcal{C}_1[\text{fix}\,(G)]]\,...]\ x_1 ... x_n \xrightarrow{\mathfrak{R}}_{+} \text{fix}\,(H)\ \ x_1 ... x_n}$$

Remark 5.1 (Fusion performs pre-generalisation). Every fusion step expresses exactly one anti-pattern to fixed-point promoted form, and generalises everything else. This technique is central to the effectiveness of this method. If we allow more than one anti-pattern into a fusion step, each could block the fusion of the other.

Some terms can be decomposed into separate anti-pattern contexts in multiple ways. The rewriting algorithm within Elea uses the numerical ordering of the rules as they are presented in this thesis to decide which rule to apply next. As the traverse rules from Section 4.4.5 on page 58 are given earlier, sub-terms will be recursed into and fused before outer terms. After this, the ordering of the fusion rules within this chapter will be used. This ordering allows Elea to resolve this ambiguity, however, experimentally I found that changing the order in which fusion rules are applied did not affect the outcome of the rewriting algorithm.

The fusion rules in this chapter each represent a particular shape of anti-pattern I have come across in examples. I expect that this set is not complete, and there may be other examples which require more fusion rules to be added to the Elea system in the future.

Remark 5.2 (Fusion vs. automated induction). The FPPF anti-patterns we have described here are exactly those things which block automated induction provers from being able to apply an inductive hypothesis within a proof. In the paper ”Productive use of failure in inductive proof” [30], the authors describe a method whereby, if a hypothesis rewrite is blocked by such an anti-pattern, this blocked term can be generalised in some way to produce a lemma which will hopefully ”unblock” the proof, viz. allowing the hypothesis to be applied.

The decomposition of multiple anti-patterns into multiple fusion contexts matches this generalisation process in automated induction, whereby each lemma generated to unblock a proof matches one or more contexts which are fused into a fixed-point. This comparison is expanded further in Section 10.2.4 on page 138.

5.2 Fusion rules

The sections after this one give a series of rules, each designed to remove a different shape of FPPF anti-pattern from a term. These rules all use the same fusion process so I decided to separate out this process as its own rewrite rule: $\mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\mathfrak{R}} \mathrm{fix}\,(H)$. All anti-pattern removing rules invoke a fusion rewrite as a antecedent in order to perform their fusion step.

Whether $\mathfrak{R}$ is $\sqsubseteq$ or $\sqsupseteq$ changes which fusion rule we can use. The rule which preserves $\sqsubseteq$ I have called truncation fusion, as its soundness is based upon the principle of truncation induction. It is for this rule that truncated fixed-points were added to the grammar of νPCF.

The rule which preserves $\sqsupseteq$ I have called ω-fusion, to contrast it with truncation fusion, but I could also have called it supercompilation, since it is functionally the same process.

Rule 21 (ω-fusion).

$$\frac{\Gamma \vdash \mathcal{C}[\mathrm{fix}\,(G)] : \tau \qquad \Gamma', \Phi', \mathcal{H}' \vdash \mathcal{C}[G\ (\mathrm{fix}\,(G))] \xrightarrow{\sqsupseteq}_{+} H}{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\sqsupseteq} \mathrm{fix}\,(\mathrm{fn}\ h : \tau.\ H)}$$

$$\begin{aligned}
\text{if}\quad & h \text{ is a fresh variable}\\
& \Gamma' = \Gamma[h \mapsto \tau]\\
& \Phi' = \Phi,\ \mathcal{C}[\mathrm{fix}\,(G)] \sqsupseteq h\\
& \mathcal{H}' = \mathcal{H},\ \mathcal{C}[\mathrm{fix}\,(G)]\\
& h \in \mathsf{freeVars}\,(H)
\end{aligned}$$

The above rule fuses a context $\mathcal{C}$ into a fixed-point $\mathrm{fix}\,(G)$, producing a new fixed-point $\mathrm{fix}\,(H)$, such that $\mathcal{C}[\mathrm{fix}\,(G)] \sqsupseteq \mathrm{fix}\,(H)$. The fact added to make Φ', $\mathcal{C}[\mathrm{fix}\,(G)] \sqsupseteq h$, means that $\mathcal{C}[\mathrm{fix}\,(G)]$ can be rewritten to h within the antecedent rewrite, using the folding rule from the next section. The check $h \in \mathsf{freeVars}\,(H)$ makes sure that this fact was used at some point, otherwise this rule will just have unrolled $\mathrm{fix}\,(G)$ in $\mathcal{C}[\mathrm{fix}\,(G)]$ and stuck a useless fix around it.

Rule 22 (truncation fusion).

$$\frac{\Gamma \vdash \mathcal{C}[\mathrm{fix}\,(G)] : \tau \qquad \Gamma', \Phi', \mathcal{H}' \vdash \mathcal{C}[G\ (\mathrm{fix}^a\,(G))] \xrightarrow{\sqsubseteq}_{+} H}{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\sqsubseteq} \mathrm{fix}\,(\mathrm{fn}\ h : \tau.\ H[\mathrm{fix}/\mathrm{fix}^a])}$$

$$\begin{aligned}
\text{if}\quad & h \text{ is a fresh variable}\\
& a \in \mathbb{N}\\
& \Gamma' = \Gamma[h \mapsto \tau]\\
& \Phi' = \Phi,\ \mathcal{C}[\mathrm{fix}^a\,(G)] \sqsubseteq h\\
& \mathcal{H}' = \mathcal{H},\ \mathcal{C}[\mathrm{fix}\,(G)]\\
& [\![\mathcal{C}[\bot]]\!] = \bot \quad (\mathcal{C} \text{ is strict})\\
& h \in \mathsf{freeVars}\,(H)
\end{aligned}$$

Truncation fusion is an adaptation of ω-fusion (supercompilation) to preserve $\sqsubseteq$ without relying on improvement theory, something discussed in Section 11.4. It fuses $\mathcal{C}$ into $\mathrm{fix}\,(G)$, yielding $\mathrm{fix}\,(H)$ such that $\mathcal{C}[\mathrm{fix}\,(G)] \sqsubseteq \mathrm{fix}\,(H)$. Unlike ω-fusion we must truncate the unrolled fixed-point $\mathrm{fix}\,(G)$ to some maximum number of unrollings, given by a new meta-variable $a \in \mathbb{N}$. The fact added to Φ', $\mathcal{C}[\mathrm{fix}^a\,(G)] \sqsubseteq h$, allows us to rewrite $\mathcal{C}[\mathrm{fix}^a\,(G)]$ to h within the antecedent rewrite using the folding rule from the next section.

Along with truncating $\mathrm{fix}\,(G)$ we must also have that $\mathcal{C}$ is strict to ensure soundness. The check $h \in \mathsf{freeVars}\,(H)$ exists for the same reason as in ω-fusion, to ensure our added fact is used at least once. The substitution $[\mathrm{fix}/\mathrm{fix}^a]$ replaces all fixed-points truncated to a with least fixed-points at the end of our rewrite. This is sound as $\mathrm{fix}^a\,(F) \sqsubseteq \mathrm{fix}\,(F)$ for any a and F.

5.2.1 Folding

Both of the fusion rules given above add a fact to our fact environment. The Rule 23 rule applies these facts as a rewrite whenever possible. I have called this rule folding, as it parallels the fold step of unfold-fold style rewrites [16].

Rule 23 (folding).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash F[A_1/x_1]...[A_n/x_n] \xrightarrow{\mathfrak{R}} h\ A_1...A_n}$$
$$\text{if} \quad ((\text{fn } x_1, ..., x_n.\, F)\ \mathfrak{R}\ h) \in \Phi$$

Recall that $\mathfrak{R}$ is a relation variable which can mean either $\sqsubseteq$ or $\sqsupseteq$, and that a rewrite $A \xrightarrow{\mathfrak{R}} B$ is sound if $A\ \mathfrak{R}\ B$. Therefore the fact $((\text{fn } x_1, ..., x_n.\, F)\ \mathfrak{R}\ h) \in \Phi$ means that we can rewrite fn $x_1, ..., x_n.\, F$ to h in this context.

Finding $((\text{fn } x_1, ..., x_n.\, F)\ \mathfrak{R}\ h) \in \Phi$ and $A_1...A_n$ are done using a term unification algorithm successively on every fact in Φ to try and unify $F[A_1/x_1]...[A_n/x_n]$ with the term on the left-hand side of the rewrite.

5.3 Constant argument fusion

Constant argument fusion is the first fusion based rewrite rule I present. It fuses a non-variable argument inside the body of a fixed-point, provided it is a constant argument for that function. Constant arguments are those which are unchanged in every recursive call to the function (Definition 5.2).

The functions in Appendix A are full of constant arguments, like the second argument of both `add` and `app`, and the first argument of `filter` and `map`.

Definition 5.2 (Constant function arguments).

$$\begin{aligned}&\mathsf{constantArgs}\,(\text{fix}^a\,(\text{fn } f, x_1, ..., x_n.\, F)) = \\ &\qquad \{\ i \mid \forall\,((f\ A_1...A_n) \in \mathsf{subterms}(F))\ .\ A_i = x_i\ \}\end{aligned}$$

constantArgs returns every function argument index which is unchanged in all recursive calls. This is used in constant-argument fusion to identify which arguments can be directly fused into the function body.

Rule 24 (constant-argument fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^a\,(G)\, A_1...A_n \xrightarrow{\mathfrak{R}} \text{fix}\,(H)\, A_1...A_{i-1}\, A_{i+1}...A_n}$$
$$\begin{aligned}\text{if} \quad & x_1...x_{n-1} \text{ are fresh variables} \\ & i \in \mathsf{constantArgs}(\text{fix}\,(G)) \\ & \mathcal{C} = \text{fn } x_1, ..., x_{n-1}.\, \Box\, x_1...x_{i-1}\, A_i\, x_i...x_n\end{aligned}$$

In this form of fusion we can remove the ith argument to a fixed-point if it is a constant argument. The context $\mathcal{C}$ expresses only that the ith argument is A_i, generalising every other argument (Remark 5.1). The required judgement ($\mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\mathfrak{R}} \mathrm{fix}\,(H)$) will invoke one of our two fusion rules (Rule 21 or Rule 22). As required by Rule 22 our context $\mathcal{C}$ is strict, since the gap is in a topmost function position.

Example 5.1. This example is a truncation fusion rewrite within a constant-argument fusion rewrite, fusing the argument $\texttt{[}y\texttt{]}$ into the body of the $\texttt{app}$ function, resulting in the $\texttt{snoc}_y$ (backwards cons) function.

The definitions of all terms are given in Appendix A, and in this example I use the list append function $\texttt{app}$, and the tail-cons function $\texttt{snoc}$, where $\texttt{app}'$ is the body of the fixed-point $\texttt{app}$, i.e. $\texttt{app} = \mathrm{fix}\,(\texttt{app}')$.

$$
\begin{array}{l}
\mathrm{fn}\ xs.\ \texttt{app}'\ \texttt{app}^a\ xs\ \texttt{[}y\texttt{]} \\
\qquad \xrightarrow{\sqsubseteq} \{\text{ by definition of } \texttt{app}' \text{ and beta reduction }\} \\
\mathrm{fn}\ xs.\ \mathrm{case}\ xs\ \mathrm{of} \\
\quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\
\quad x :: xs' \to x :: \texttt{app}^a\ xs'\ \texttt{[}y\texttt{]} \\
\qquad \xrightarrow{\sqsubseteq} \{\text{ by folding } \mathrm{fn}\ xs.\ \texttt{app}^a\ xs\ \texttt{[}y\texttt{]} \sqsubseteq h \ \} \\
\mathrm{fn}\ xs.\ \mathrm{case}\ xs\ \mathrm{of} \\
\quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\
\quad x :: xs' \to x :: h\ xs' \\
\hline
(\mathrm{fn}\ xs.\ \square\ xs\ \texttt{[}y\texttt{]}) \oplus \texttt{app} \xrightarrow{\sqsubseteq} \mathrm{fix} \left(\begin{array}{l} \mathrm{fn}\ h, xs.\ \mathrm{case}\ xs\ \mathrm{of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to x :: h\ xs' \end{array} \right) \quad (=_\alpha \texttt{snoc}_y)
\end{array}
$$

In the above example, the rewrite given above the derivation line is:

$$
\mathrm{fn}\ xs.\ \texttt{app}'\ \texttt{app}^a\ xs\ \texttt{[}y\texttt{]} \quad \xrightarrow{\sqsubseteq}_+ \quad \begin{array}{l} \mathrm{fn}\ xs.\ \mathrm{case}\ xs\ \mathrm{of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to x :: h\ xs' \end{array}
$$

This corresponds to the rewrite above the derivation line in the truncation fusion rule: $\mathcal{C}[G\ (\mathrm{fix}^a\,(G))] \xrightarrow{\sqsubseteq}_+ H$. The rewrite below the derivation line is the consequence of applying the truncation fusion rule.

Example 5.2. This example is a truncation fusion rewrite within a constant-argument fusion rewrite, fusing the argument $\texttt{Suc}\ y$ into the body of the $\texttt{add}$ function, resulting in the $\texttt{add}_{(\texttt{Suc}\ y)}$ function.

The definitions of all terms are given in Appendix A, and in this example I use the addition function $\texttt{add}$, where $\texttt{add}'$ is the body of the fixed-point $\texttt{add}$, i.e. $\texttt{add} = \text{fix}\,(\texttt{add}')$.

$$
\begin{array}{l}
\text{fn } x.\, \texttt{add}'\, \texttt{add}^a\, x\, (\texttt{Suc}\, y) \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by definition of } \texttt{add}' \text{ and beta reduction } \} \\
\text{fn } x.\, \text{case } x \text{ of} \\
\quad \texttt{0} \to \texttt{Suc}\, y \\
\quad \texttt{Suc}\, x' \to \texttt{Suc}\, (\texttt{add}^a\, x'\, (\texttt{Suc}\, y)) \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by folding fn } x.\, \texttt{add}^a\, x\, (\texttt{Suc}\, y) \sqsubseteq h \ \} \\
\text{fn } x.\, \text{case } x \text{ of} \\
\quad \texttt{0} \to \texttt{Suc}\, y \\
\quad \texttt{Suc}\, x' \to \texttt{Suc}\, (h\, x') \\
\hline
(\text{fn } x.\, \Box\, x\, (\texttt{Suc}\, y)) \oplus \texttt{add} \xrightarrow{\sqsubseteq} \text{fix} \left(\begin{array}{l} \text{fn } h, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\, y \\ \quad \texttt{Suc}\, x' \to \texttt{Suc}\, (h\, x') \end{array} \right) \quad (=_\alpha \texttt{add}_{(\texttt{Suc}\, y)})
\end{array}
$$

In the above example, the rewrite given above the derivation line is:

$$
\texttt{add}'\, \texttt{add}^a\, x\, (\texttt{Suc}\, y) \quad \xrightarrow{\sqsubseteq}_{+} \quad \begin{array}{l} \text{fn } x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\, y \\ \quad \texttt{Suc}\, x' \to \texttt{Suc}\, (h\, x') \end{array}
$$

This corresponds to the rewrite above the derivation line in the truncation fusion rule: $\mathcal{C}[G\ (\text{fix}^a\ (G))] \xrightarrow{\sqsubseteq}_{+} H$. The rewrite below the derivation line is the consequence of applying the truncation fusion rule, which gives us:

$$
(\text{fn } x.\, \Box\, x\, (\texttt{Suc}\, y)) \oplus \texttt{add} \xrightarrow{\sqsubseteq} \texttt{add}_{(\texttt{Suc}\, y)}
$$

Using a rewrite from the next chapter (constructor fission) we can then do

$$
\texttt{add}_{(\texttt{Suc}\, y)}\, x \xrightarrow{\sqsubseteq} \texttt{Suc}\, (\texttt{add}_y\, x)
$$

This, combined with the above fusion step, gives us the full rewrite

$$
\texttt{add}\, x\, (\texttt{Suc}\, y) \xrightarrow{\sqsubseteq}_{+} \texttt{Suc}\, (\texttt{add}_y\, x)
$$

5.4 Fusing a fixed-point into a fixed-point

Fixed-point promoted form requires that all arguments to fixed-points be variables. Fix-fix fusion takes a fixed-point which has another fixed-point as an argument and fuses the topmost fixed-point into the argument fixed-point, bringing us closer to having all variable arguments.

Rule 25 (fix-fix fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(F)\, A_1...A_{i-1}\, (\text{fix}\,(G)\, A_i...A_j)\;\; A_{j+1}...A_n \xrightarrow{\mathfrak{R}} \text{fix}\,(H)\;\; A_1...A_n}$$

if $x_1...x_n$ are fresh variables

such that $\forall(i, j \in \mathbb{N})\;.\; A_i = A_j \Rightarrow x_i = x_j$

$\mathcal{C} = \text{fn}\; x_1, ..., x_n.\; \text{fix}\,(F)\, x_1...x_{i-1}\; (\Box\; x_i...x_j)\; x_{j+1}...x_n$

$i \in \mathsf{strictArgs}(\text{fix}\,(G))$

Here I define the fusion of a fixed-point into its ith argument. The strictness requirement is to fulfil the strictness requirement of Rule 22. The context $\mathcal{C}$ expresses only the application of the outer fixed-point, preserving only when arguments are equivalent, and generalising away all other arguments.

You may recall Remark 5.1 on page 66 stated that fusion steps generalise away all anti-patterns except one. As you can see this was not entirely true, as this step preserves when arguments are equivalent. I found experimentally that when performing fix-fix fusion, and constructor fusion given later, one needs to preserve argument equivalence in the context being fused, but every other feature could be generalised away.

Example 5.3. This example is a truncation fusion rewrite within a fix-fix fusion rewrite, fusing the `rev` function into the $\texttt{snoc}_y$ function.

The definitions of all terms are given in Appendix A, and in this example I use the list reversal function `rev`, and the tail-cons function `snoc`, where $\texttt{rev}'$ is the body of the fixed-point `rev`, i.e. $\texttt{rev} = \text{fix}\,(\texttt{rev}')$.

$\text{fn}\; xs.\; \texttt{rev}\; (\texttt{snoc}'_y\; \texttt{snoc}^a_y\; xs)$

$\xrightarrow{\sqsubseteq}$ { by definition of $\texttt{snoc}'_y$ and beta reduction }

$$\text{fn}\; xs.\; \texttt{rev} \begin{pmatrix} \text{case}\; xs\; \text{of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x \texttt{ :: } xs' \to x \texttt{ :: } \texttt{snoc}^a_y\; xs' \end{pmatrix}$$

$\xrightarrow{\sqsubseteq}$ { by float apply-case }

$\text{fn}\; xs.\; \text{case}\; xs\; \text{of}$
  $\texttt{[ ]} \to \texttt{rev [}y\texttt{]}$
  $x \texttt{ :: } xs' \to \texttt{rev}\; (x \texttt{ :: } \texttt{snoc}^a_y\; xs')$

$\xrightarrow{\sqsubseteq}_+$ { by unfold fix twice }

$\text{fn}\; xs.\; \text{case}\; xs\; \text{of}$
  $\texttt{[ ]} \to \texttt{[}y\texttt{]}$
  $x \texttt{ :: } xs' \to \texttt{snoc}_x\; (\texttt{rev}\; (\texttt{snoc}^a_y\; xs'))$

$\xrightarrow{\sqsubseteq}$ { by folding fn xs. $\texttt{rev}$ $(\texttt{snoc}^a_y\, xs) \sqsubseteq h$ }

fn xs. case xs of
  $\texttt{[ ]} \to \texttt{[}y\texttt{]}$
  $x :: xs' \to \texttt{snoc}_x\,(h\, xs')$

$$\frac{}{\text{fn } xs.\ \texttt{rev}\ (\square\, xs) \oplus \texttt{snoc}_y \xrightarrow{\sqsubseteq} \text{fix} \begin{pmatrix} \text{fn } h, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \texttt{snoc}_x\,(h\, xs') \end{pmatrix}}$$

In the above example, the rewrite given above the derivation line is:

$$\text{fn } xs.\ \texttt{rev}\ (\texttt{snoc}'_y\ \texttt{snoc}^a_y\, xs) \quad \xrightarrow{\sqsubseteq}_+ \quad \begin{array}{l} \text{fn } xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \texttt{snoc}_x\,(h\, xs') \end{array}$$

This corresponds to the rewrite above the derivation line in the truncation fusion rule: $\mathcal{C}[G\ (\text{fix}^a\,(G))] \xrightarrow{\sqsubseteq}_+ H$. The rewrite below the derivation line is the consequence of applying the truncation fusion rule.

Example 5.4. This example is a truncation fusion rewrite within a fix-fix fusion rewrite, which fuses the `rev` function into itself, resulting in a recursive identity function. One antecedent rewrite refers to Example 6.7, which is one of the fission rules we introduce in the next chapter.

fn xs. $\texttt{rev}\ (\texttt{rev}'\ \texttt{rev}^a\, xs)$

$\xrightarrow{\sqsubseteq}$ { by definition of $\texttt{rev}'$ and beta reduction }

$$\text{fn } xs.\ \texttt{rev} \begin{pmatrix} \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[ ]} \\ \quad y :: ys \to \texttt{snoc}_y\,(\texttt{rev}^a\, ys) \end{pmatrix}$$

$\xrightarrow{\sqsubseteq}$ { by case-app }

fn xs. case xs of
  $\texttt{[ ]} \to \texttt{rev [ ]}$
  $y :: ys \to \texttt{rev}\,(\texttt{snoc}_y\,(\texttt{rev}^a\, ys))$

$\xrightarrow{\sqsubseteq}$ { by unfold fix }

fn xs. case xs of
  $\texttt{[ ]} \to \texttt{[ ]}$
  $y :: ys \to \texttt{rev}\,(\texttt{snoc}_y\,(\texttt{rev}^a\, ys))$

$\xrightarrow{\sqsubseteq}_+$ { by Example 6.7 on page 94 }

fn xs. case xs of
  $\texttt{[ ]} \to \texttt{[ ]}$
  $y :: ys \to y :: \texttt{rev}\,(\texttt{rev}^a\, ys)$

$\xrightarrow{\sqsubseteq}$ { by folding $\texttt{rev}\ (\texttt{rev}^a\ xs) \sqsubseteq h$ }

$$\begin{array}{l}\text{fn } xs.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{[ ]}\\ \quad y :: ys \to y :: h\, ys\end{array}$$

$$\dfrac{}{\text{fn } xs.\ \texttt{rev}\ (\Box\ xs) \oplus \texttt{rev} \xrightarrow{\sqsubseteq} \text{fix}\left(\begin{array}{l}\text{fn } h, xs.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{[ ]}\\ \quad y :: ys \to y :: h\, ys\end{array}\right)}$$

In the above example, the rewrite given above the derivation line is:

$$\text{fn } xs.\ \texttt{rev}\ (\texttt{rev}'\ \texttt{rev}^a\ xs) \quad \xrightarrow{\sqsubseteq}_+ \quad \begin{array}{l}\text{fn } xs.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{[ ]}\\ \quad y :: ys \to y :: h\, ys\end{array}$$

This corresponds to the rewrite above the derivation line in the truncation fusion rule: $\mathcal{C}[G\ (\text{fix}^a\ (G))] \xrightarrow{\sqsubseteq}_+ H$. The rewrite below the derivation line is the consequence of applying the truncation fusion rule. The result of applying this truncation fusion as the antecedent fusion rewrite within fix-fix fusion gives us:

$$\text{fn } xs.\ \texttt{rev}\ (\texttt{rev}\ xs) \xrightarrow{\sqsubseteq} \text{fix}\left(\begin{array}{l}\text{fn } h, xs.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{[ ]}\\ \quad y :: ys \to y :: h\, ys\end{array}\right)$$

Using a rewrite from the next chapter (identity fission) we can then do:

$$\text{fix}\left(\begin{array}{l}\text{fn } h, xs.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{[ ]}\\ \quad y :: ys \to y :: h\, ys\end{array}\right) \xrightarrow{\sqsubseteq} \text{fn } xs.\ xs$$

This, combined with the above fusion step, allows our system to rewrite:

$$\texttt{rev}\ (\texttt{rev}\ xs) \xrightarrow{\sqsubseteq}_+ xs$$

Example 5.5. This is an example of using fix-fix fusion on a corecursive fixed-point, namely the fusion of the the $\texttt{map}_f$ function into the fixed-point $\texttt{repeat}_x$, resulting in a fixed-point alpha equal to $\texttt{repeat}_{(f\,x)}$.

$\texttt{map}_f\ (\texttt{repeat}'_x\ \texttt{repeat}^a_x)$

$\xrightarrow{\sqsubseteq}$ { by definition of $\texttt{repeat}'$ }

$\texttt{map}_f\ (x :: \texttt{repeat}^a_x)$

$\xrightarrow{\sqsubseteq}$ { by unfold-fix }

$f\,x :: \texttt{map}_f\ \texttt{repeat}^a_x$

$\xrightarrow{\sqsubseteq}$ { by folding $\texttt{map}_f\ \texttt{repeat}^a_x \sqsubseteq h$ }

$f\,x :: h$

$$\dfrac{}{\texttt{map}_f\ \Box \oplus \texttt{repeat}_x \xrightarrow{\sqsubseteq} \text{fix}\,(\text{fn } h.\ f\,x :: h) \qquad (=_\alpha \texttt{repeat}_{(f\,x)})}$$

5.5 Constructor fusion

Another anti-pattern to FPPF is a constructor as an argument to a fixed-point. In most cases a constructor argument requires the fixed-point to be unfolded, but there are examples of when this is not the approach to take. One such case is $\texttt{eq}\, x\, (\texttt{Suc}\, x)$, where $\texttt{eq}$ is the equality predicate on $\texttt{Nat}$. In this case Elea would instead use constructor fusion to fuse the $\text{fn}\ x.\, \Box\, x\, (\texttt{Suc}\, x)$ context into the $\texttt{eq}$ fixed-point, yielding a fixed-point which either returns $\texttt{False}$ or is undefined (if $x \sqsubseteq \texttt{Suc}\, x$).

Rule 26 (constructor fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(G)\, A_1 ... A_{i-1}\, (\text{con}_k \langle \boldsymbol{T} \rangle\, A_i ... A_j)\, A_{j+1} ... A_n \xrightarrow{\mathfrak{R}} \text{fix}\,(H)\, A_1 ... A_n}$$

$$\begin{aligned}
\text{if}\quad & x_1, ..., x_n \text{ fresh variables} \\
& \qquad \text{such that } \forall (i, j \in \mathbb{N})\ .\ A_i = A_j \Rightarrow x_i = x_j \\
& \mathcal{C} = \text{fn}\ x_1, ..., x_n.\, \Box\ x_1 ... x_{i-1}\, (\text{con}_k \langle \boldsymbol{T} \rangle\, x_i ... x_j)\, x_j ... x_n
\end{aligned}$$

This rule fuses a constructor argument into the body of a fixed-point itself. As with fix-fix fusion the only information which not generalised is the existence of any repeated arguments, something discussed earlier in Rule 25 (fix-fix fusion). As with all fusion rules we require the context $\mathcal{C}$ to be strict in its gap, which is always the case here as the gap appears in a topmost function position.

This rule is applicable to the same shape of term as the unfold-fix rule given earlier in Rule 19. The unfold-fix rule has a higher precedence, as it comes earlier in the rule numbering, so Elea will attempt to apply it first. Only if that rule has failed its post-hoc embedding check will Elea fall through to this rule. This is to say that Elea will always attempt to unfold a fixed-point before performing constructor fusion.

Example 5.6. This example is a truncation fusion rewrite inside a constructor fusion rewrite which transforms the term $\text{fn}\ x.\, \texttt{eq}\, x\, (\texttt{Suc}\, x)$ into just $\text{fn}\ x.\, \texttt{False}$.

$\text{fn}\ x.\, \texttt{eq}'\, \texttt{eq}^a\, x\, (\texttt{Suc}\, x)$

$\xrightarrow{\sqsubseteq}$ { by definition of $\texttt{eq}'$ and beta reduction }

$\text{fn}\ x.\, \text{case}\ x, \texttt{Suc}\, x\ \text{of}$

$$\begin{array}{lll}
\texttt{0}, \texttt{0} & \rightarrow & \texttt{True} \\
\texttt{0}, \texttt{Suc}\, y' & \rightarrow & \texttt{False} \\
\texttt{Suc}\, x', \texttt{0} & \rightarrow & \texttt{False} \\
\texttt{Suc}\, x', \texttt{Suc}\, y' & \rightarrow & \texttt{eq}^a\, x'\, y'
\end{array}$$

$\xrightarrow{\sqsubseteq}$ { by case-var substitution and case reduction }

$\text{fn}\ x.\, \text{case}\ x\ \text{of}$

$\texttt{0} \rightarrow \texttt{False}$

$\texttt{Suc}\, x' \rightarrow \texttt{eq}^a\, x'\, (\texttt{Suc}\, x')$

$$\xrightarrow{\sqsubseteq}_{+} \{ \text{ by folding } (\text{fn } x.\, \texttt{eq}^{a}\; x\; (\texttt{Suc}\; x) \sqsubseteq h) \; \}$$

$$\begin{array}{l}\text{fn } x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{False} \\ \quad \texttt{Suc}\; x' \to h\; x'\end{array}$$

$$\overline{\quad\text{fn } x.\, \square\; x\; (\texttt{Suc}\; x) \oplus \texttt{eq} \xrightarrow{\sqsubseteq} \text{fix}\begin{pmatrix}\text{fn } h, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{False} \\ \quad \texttt{Suc}\; x' \to h\; x'\end{pmatrix}\quad}$$

Using the sub-term fission rewrite from the next chapter we have:

$$\text{fix}\begin{pmatrix}\text{fn } h, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{False} \\ \quad \texttt{Suc}\; x' \to h\; x'\end{pmatrix} \xrightarrow{\sqsubseteq} \text{fn } x.\, \texttt{False}$$

Combined with the above constructor fusion step our tool can rewrite:

$$\texttt{eq}\; x\; (\texttt{Suc}\; x) \xrightarrow{\sqsubseteq}_{+} \texttt{True}$$

5.6 Repeated variable fusion

One feature of fixed-point promoted form is that all of the variable arguments to a fixed-point are unique. If Elea encounters variables which occur more than once as the argument to a fixed-point it can use repeated-variable fusion to remove the repetition.

Rule 27 (repeated-variable fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(G)\; A_1 ... A_n \xrightarrow{\mathfrak{R}} \text{fix}\,(H)\; A_1 ... A_n}$$

$$\begin{array}{ll}\text{if} & x_1, ..., x_n \text{ fresh variables} \\ & \qquad \text{such that } \forall (i, j \in \mathbb{N})\; \exists y \,.\, y = A_i = A_j \Rightarrow x_i = x_j \\ & \mathcal{C} = \text{fn } x_1, ..., x_n.\, \square\; x_1 ... x_n\end{array}$$

This rule fuses repeated variable arguments into a fixed-point, such that the resulting fixed-point has only unique arguments. As with other fusion rules the context $\mathcal{C}$ must be strict, which is always the case in this rule as the gap occurs as a topmost function call. The repeated arguments still exist in the resulting fixed-point call, but will be removed by constant argument fusion after this step (see the end of Example 5.7).

Example 5.7. This example is a truncation fusion rewrite within a repeated-variables fusion rewrite, in which we rewrite the term $\texttt{lq}\; x\; x$ into a fixed-point which either returns

$\texttt{True}$, or is undefined.

$$\begin{array}{l}
\text{fn } x, x.\, \texttt{lq}' \, \texttt{lq}^a \, x \, x \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by definition of } \texttt{lq}' \text{ and beta reduction } \} \\
\text{fn } x.\, \text{case } x, x \text{ of} \\
\quad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \texttt{True} \\
\texttt{0}, \texttt{Suc}\, y' & \rightarrow & \texttt{True} \\
\texttt{Suc}\, x', \texttt{0} & \rightarrow & \texttt{False} \\
\texttt{Suc}\, x', \texttt{Suc}\, y' & \rightarrow & \texttt{lq}^a \, x' \, y'
\end{array} \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by case-var substitution and case reduction } \} \\
\text{fn } x.\, \text{case } x \text{ of} \\
\quad \texttt{0} \rightarrow \texttt{True} \\
\quad \texttt{Suc}\, x' \rightarrow \texttt{lq}^a \, x' \, x' \\
\qquad \xrightarrow{\sqsubseteq}_{+} \{ \text{ by folding fn } x, x.\, \texttt{lq}^a \, x \, x \sqsubseteq h \ \} \\
\text{fn } x.\, \text{case } x \text{ of} \\
\quad \texttt{0} \rightarrow \texttt{True} \\
\quad \texttt{Suc}\, x' \rightarrow h \perp x'
\end{array}$$

$$\frac{}{\text{fn } x, x.\, \square \, x \, x \oplus \texttt{lq} \xrightarrow{\sqsubseteq} \text{fix} \begin{pmatrix} \text{fn } h, x, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \rightarrow \texttt{True} \\ \quad \texttt{Suc}\, x' \rightarrow h \perp x' \end{pmatrix}}$$

In the folding step above the value to the first argument can be anything, so I chose $\perp$ as it makes for a good placeholder. The first argument to the resulting fixed-point is hence no longer used within the fixed-point body, and so constant argument fusion will remove it. After this we can apply the subterm fission step from Section 6.2 to yield $\texttt{True}$.

$$\begin{array}{l}
\text{fix} \begin{pmatrix} \text{fn } h, x, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \rightarrow \texttt{True} \\ \quad \texttt{Suc}\, x' \rightarrow h \perp x' \end{pmatrix} x \, x \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by constant argument fusion } \} \\
\text{fix} \begin{pmatrix} \text{fn } h, x.\, \text{case } x \text{ of} \\ \quad \texttt{0} \rightarrow \texttt{True} \\ \quad \texttt{Suc}\, x' \rightarrow h \, x' \end{pmatrix} x \\
\qquad \xrightarrow{\sqsubseteq} \{ \text{ by sub-term fission } \} \\
\texttt{True}
\end{array}$$

Combined with the above repeated-variable fusion step our tool can rewrite:

$$\texttt{lq}\, x \, x \xrightarrow{\sqsubseteq}_{+} \texttt{True}$$

5.7 Free variable fusion

Another feature of fixed-point promoted form is that every variable argument to a fixed-point not occur freely within the fixed-point. The rule free-variable fusion removes this anti-pattern to FPPF.

Rule 28 (free-variable fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \mathrm{fix}\,(G) \xrightarrow{\mathfrak{R}} \mathrm{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \mathrm{fix}\,(G)\; A_1 ... A_n \xrightarrow{\mathfrak{R}} \mathrm{fix}\,(H)\; A_1 ... A_n}$$

$$\begin{aligned} \text{if}\quad & y_1, ..., y_{n-1} \text{ fresh variables} \\ & A_i = x \\ & x \in \mathsf{freeVars}\,(G) \\ & \mathcal{C} = \mathrm{fn}\ y_1, ..., y_{i-1}, x, y_i, ..., y_{n-1}.\, \square\ y_1 ... y_{i-1}\ x\ y_i ... y_{n-1} \end{aligned}$$

This step fuses a free variable into a fixed-point if that variable also occurs as an argument to that fixed-point. The variables $y_1, ..., y_{n-1}$ generalise every other argument.

Example 5.8. This example is a truncation fusion rewrite within a free-variable fusion rewrite, fusing the free x argument into the $\mathtt{add}_x$ function, yielding a function α-equal to `double`.

$$\begin{aligned}
& \mathrm{fn}\ x.\ \mathtt{add}'_x\ \mathtt{add}^a_x\ x \\
& \qquad \xrightarrow{\sqsubseteq} \{ \text{ by definition of } \mathtt{add}'_x \text{ and beta reduction } \} \\
& \mathrm{fn}\ x.\ \mathrm{case}\ x\ \mathrm{of} \\
& \quad \mathtt{0} \to x \\
& \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{add}^a_x\ x') \\
& \qquad \xrightarrow{\sqsubseteq} \{ \text{ by case-var substitution } \} \\
& \mathrm{fn}\ x.\ \mathrm{case}\ x\ \mathrm{of} \\
& \quad \mathtt{0} \to \mathtt{0} \\
& \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{add}^a_{(\mathtt{Suc}\ x')}\ x') \\
& \qquad \xrightarrow{\sqsubseteq} \{ \text{ by constructor fission (given later on page 92) } \} \\
& \mathrm{fn}\ x.\ \mathrm{case}\ x\ \mathrm{of} \\
& \quad \mathtt{0} \to \mathtt{0} \\
& \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{Suc}\ (\mathtt{add}^a_{x'}\ x')) \\
& \qquad \xrightarrow{\sqsubseteq} \{ \text{ by folding } \mathrm{fn}\ x.\ \mathtt{add}_x\ x \sqsubseteq h \ \} \\
& \mathrm{fn}\ x.\ \mathrm{case}\ x\ \mathrm{of} \\
& \quad \mathtt{0} \to \mathtt{0} \\
& \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{Suc}\ (h\ x'))
\end{aligned}$$

$$\overline{\mathrm{fn}\ x.\ \square\ x \oplus \mathtt{add}_x \xrightarrow{\sqsubseteq} \mathrm{fix} \begin{pmatrix} \mathrm{fn}\ h, x.\ \mathrm{case}\ x\ \mathrm{of} \\ \quad \mathtt{0} \to \mathtt{0} \\ \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (\mathtt{Suc}\ (h\ x')) \end{pmatrix} \qquad (=_\alpha \mathtt{double})}$$

5.8 Accumulation fusion

Some arguments to fixed-points are known as accumulating arguments. These are values which are in some way increased in recursive calls within the fixed-point. Below I give the definition of a list reversal function `it-rev` in which the second argument is accumulating:

$$\texttt{it-rev} \overset{\text{def}}{=} \text{fix}\begin{pmatrix}\text{fn } f, xs, ys.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \rightarrow ys \\ \quad x :: xs' \rightarrow f\ xs'\ (x :: ys)\end{pmatrix}$$

It is the case that `rev` $xs \cong$ `it-rev` xs `[ ]`, however, `rev` xs is in fixed-point promoted form, but `it-rev` xs `[ ]` is not, as the second argument is not a variable. For terms like this Elea uses the accumulation fusion rewrite given in this chapter, which in this case would rewrite `it-rev` xs `[ ]` into `rev` xs. Elea does not require the definition of `rev` in order to perform this rewrite, as it invents the definition in the course of rewriting `it-rev` xs `[ ]`.

The operator Definition 5.3 defines how we can detect which arguments to a function are accumulating, and Rule 29 gives the accumulation fusion rule. Example 5.9 shows how we could use this rule to rewrite `it-rev` xs `[ ]` $\xrightarrow{\sqsubseteq}_{+}$ `rev` xs, though most of the heavy lifting in this rewrite is done by the fold discovery technique from Chapter 8.

Definition 5.3 ($\mathsf{accumulatingArgs}$).

$$\begin{aligned}&\mathsf{accumulatingArgs}(\text{fix}^{a}\,(\text{fn } f, x_1, ..., x_n.\ F)) \overset{\text{def}}{=} \\ &\quad \{\ i \ \mid\ \exists (f\ A_1 ... A_n \in \mathsf{subterms}(F))\ . \\ &\qquad\qquad A_i \text{ is not } x_i \text{ or a subterm of a pattern } x_i \text{ has been matched to }\}\end{aligned}$$

The $\mathsf{accumulatingArgs}$ operator returns the argument indices of a fixed-point that cannot be shown decrease or stay constant in every recursive call.

Rule 29 (accumulation fusion).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^{a}\,(G)\ A_1 ... A_n \xrightarrow{\mathfrak{R}} \text{fix}\,(H)\ A_1 ... A_{i-1}\ A_{i+1} ... A_n}$$

$$\begin{aligned}\text{if}\quad & x_1 ... x_{n-1} \text{ are fresh variables} \\ & A_i \text{ is not a variable, and contains no fixed-points} \\ & i \in \mathsf{accumulatingArgs}(\text{fix}\,(G)) \\ & \mathcal{C} = \text{fn } x_1, ..., x_{n-1}.\ \square\ x_1 ... x_{i-1}\ A_i\ x_i ... x_n\end{aligned}$$

This rule fuses an accumulating argument into a fixed-point, attempting to remove it. This rule will rewrite `it-rev` xs `[ ]` $\xrightarrow{\sqsubseteq}$ `rev` xs (Example 5.9). As required by the fusion rewrite our context $\mathcal{C}$ is strict, since the gap is in a topmost function position.

Example 5.9. This example is a truncation fusion rewrite within a accumulation fusion rewrite, which rewrites the term fn xs. `it-rev` xs `[ ]` into `rev`. All terms are defined in Appendix A. One antecedent rewrite refers to fold discovery, a technique described in Chapter 8. It is actually the case that fold discovery performs the folding rewrite within itself, but here I present them as two separate steps for clarity. This is to say that my tool does the second and third antecedent rewrite presented here as one single rewrite.

$$\text{fn } xs.\, \texttt{it-rev}'\ \texttt{it-rev}^a\ xs\ \texttt{[ ]}$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by definition of } \texttt{it-rev}' \text{ and beta reduction } \}$$

$$\begin{array}{l} \text{fn } xs.\, \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \rightarrow \texttt{[ ]} \\ \quad x :: xs' \rightarrow \texttt{it-rev}^a\ xs'\ [x] \end{array}$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by fold discovery } \}$$

$$\begin{array}{l} \text{fn } xs.\, \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \rightarrow \texttt{[ ]} \\ \quad x :: xs' \rightarrow \texttt{snoc}_x\ (\texttt{it-rev}^a\ xs'\ \texttt{[ ]}) \end{array}$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by folding fn } xs.\, \texttt{it-rev}^a\ xs\ \texttt{[ ]} \sqsubseteq h \}$$

$$\begin{array}{l} \text{fn } xs.\, \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \rightarrow \texttt{[ ]} \\ \quad y :: ys \rightarrow \texttt{snoc}_x\ (h\ xs') \end{array}$$

$$\overline{\text{fn } xs.\, \Box\, xs\ \texttt{[ ]} \oplus \texttt{it-rev} \xrightarrow{\sqsubseteq} \text{fix} \begin{pmatrix} \text{fn } h, xs.\, \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \rightarrow \texttt{[ ]} \\ \quad x :: xs' \rightarrow \texttt{snoc}_x\ (h\ xs') \end{pmatrix} \qquad (=_\alpha \texttt{rev})}$$

5.9 Fact fusion

In order to compete with automated induction provers, and prove properties such as `sorted` (`isort` xs) $\sqsubseteq$ `True`, Elea needs to be able to reason about unsatisfiability. For example, in the proof of the aforementioned property the prover Zeno [64] shows the following term to be unsatisfiable, which is to say it proves there is no value of x and y such that it holds.

$$\texttt{lq}\, x\, y \cong \texttt{False} \ \wedge\ \texttt{lq}\, y\, x \cong \texttt{False}$$

In this section I describe fact fusion, a proof technique for showing unsatisfiability of branches within fixed-points. I first detail fact fusion at a high level to give some insight into how we can use fusion to remove unsatisifiable fixed-point branches. I then give the fact fusion rewrite rule, followed by an auxiliary rules, sink assert, which is pivotal to the success of fact fusion. This section ends with two examples of fact fusion, which show how Elea uses it to automatically perform the following rewrites.

$$\texttt{or}\ (\texttt{lq}\, x\, y)\ (\texttt{lq}\, y\, x) \xrightarrow{\sqsubseteq}_+ \texttt{or}\ (\texttt{lq}\, x\, y)\ \texttt{True}$$

$$\texttt{or}\ (\texttt{elem}_n\ xs)\ (\texttt{elem}_n\ (\texttt{snoc}_y\ xs)) \xrightarrow{\sqsubseteq}_+ \texttt{or}\ (\texttt{elem}_n\ xs)\ (\texttt{eq}\, n\, y)$$

The second example above expresses that if we know n is not an element of the list xs then we can rewrite $\texttt{elem}_n\ (\texttt{snoc}_y\ xs)$ to $\texttt{eq}\, n\, y$, viz checking whether n is an element of the list xs with y appended to the end approximates just checking whether n equals y.

At a high-level, fact fusion replaces unreachable branches in a fixed-point body with $\bot$, where this unreachability is inferred from the pattern matches stored within our fact environment Φ. For example, if we have $(\texttt{False} \sqsubseteq \texttt{lq}\, x\, y) \in \Phi$ we know that in this environment the number x is greater than the number y, hence if we encounter the term $\texttt{lq}\, y\, x$ we know that it will only be given values of x greater than y. We can use this to replace any return branches within the fixed-point $\texttt{lq}$ in which x is not greater than y with the term $\bot$, shown below.

$$
\begin{aligned}
&\texttt{lq}\, y\, x \\
&\quad = \{ \text{ by definition of } \texttt{lq} \ \} \\
&\text{fix} \begin{pmatrix} \text{fn } f, y, x.\ \text{case } y, x \text{ of} & & \\ \quad \texttt{0}, \texttt{0} & \to & \texttt{True} \\ \quad \texttt{0}, \texttt{Suc}\, x' & \to & \texttt{True} \\ \quad \texttt{Suc}\, y', \texttt{0} & \to & \texttt{False} \\ \quad \texttt{Suc}\, y', \texttt{Suc}\, x' & \to & f\ y'\ x' \end{pmatrix} y\, x \\
&\quad \xrightarrow{\sqsubseteq} \{ \text{ by fact fusion given } (\texttt{False} \sqsubseteq \texttt{lq}\, x\, y) \in \Phi \ \} \\
&\text{fix} \begin{pmatrix} \text{fn } f, y, x.\ \text{case } y, x \text{ of} & & \\ \quad \texttt{0}, \texttt{0} & \to & \bot \\ \quad \texttt{0}, \texttt{Suc}\, x' & \to & \texttt{True} \\ \quad \texttt{Suc}\, y', \texttt{0} & \to & \bot \\ \quad \texttt{Suc}\, y', \texttt{Suc}\, x' & \to & f\ y'\ x' \end{pmatrix} y\, x \\
&\quad \xrightarrow{\sqsubseteq} \{ \text{ by subterm fission (page 89) } \} \\
&\texttt{True}
\end{aligned}
$$

But how does this work? Recall that our truncation fusion rule takes a context $\mathcal{C}$ and a least fixed-point $\text{fix}\,(G)$, and fuses the context into the fixed-point, yielding a fixed-point $\text{fix}\,(H)$ such that $\mathcal{C}[\text{fix}\,(G)] \sqsubseteq \text{fix}\,(H)$. Fact fusion uses this fusion rule to fuse a pattern match from our fact environment Φ into a fixed-point. We can express a pattern match as a context using Lemma 5.1.

Lemma 5.1 (Asserting facts). Given terms M and A, pattern term p, type environment Γ and Γ-environment ρ.

$$
\begin{aligned}
&\text{if} && [\![p]\!]\,\rho \sqsubseteq [\![M]\!]\,\rho \\
&\text{then} && [\![A]\!]\,\rho = [\![\text{assert } M \leftarrow p \text{ in } A]\!]\,\rho
\end{aligned}
$$

The above lemma states that if we have matched term M to pattern p in this environment, then any term A is equivalent to assert $M \leftarrow p$ in A. The assert $M \leftarrow p$ in A syntax is given in Definition 2.17 on page 24 and represents a pattern match on M which returns A if M matches p, and $\bot$ otherwise. So, in the above example, our fixed-point is `lq`, and the context we fuse into it is $\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \Box\, y\, x$. This example expressed in as a truncation fusion rewrite is given below, and as a fully worked example along with its antecedent rewrite in Example 5.10.

$$
\begin{aligned}
&(\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \Box\, y\, x) \oplus \texttt{lq} \\
&\qquad \xrightarrow{\sqsubseteq} \{ \text{ by truncation fusion } \} \\
&\text{fix} \begin{pmatrix} \text{fn } f, y, x.\ \text{case } y, x \text{ of} & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \bot \\ \quad \texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\ \quad \texttt{Suc}\, y', \texttt{0} & \rightarrow & \bot \\ \quad \texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & f\, y'\, x' \end{pmatrix}
\end{aligned}
$$

Now that fact fusion has been described at a high-level, I give the rewrite rule itself, which Elea will automatically apply whenever possible, along with an auxiliary rewrite rule: sink assert.

Rule 30 (fact fusion).

$$
\frac{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\sqsubseteq} \text{fix}\,(H)}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(F)\, x_1 ... x_n \xrightarrow{\sqsubseteq} \text{fix}\,(H)\, z_1 ... z_k}
$$

$$
\begin{aligned}
\text{if}\quad & (p \sqsubseteq \text{fix}\,(G)\, y_1 ... y_m) \in \Phi \\
& \{\, z_1, ..., z_k \,\} = \{\, x_1, ..., x_n, y_1, ..., y_m \,\} \\
& k < n + m \\
& \mathcal{C} = \text{fn } z_1, ..., z_k.\ \text{assert } p \leftarrow \text{fix}\,(G)\, y_1 ... y_m \text{ in } \Box\, x_1 ... x_n
\end{aligned}
$$

This step fuses the fact that $\text{fix}\,(G)\, y_1 ... y_m$ is approximated by pattern p, into the fixed-point $\text{fix}\,(F)\, x_1 ... x_n$. The context $\mathcal{C}$ expresses this fact using our assertion syntax from page 24.

$\{\, z_1, ..., z_k \,\}$ are all of the variables from $x_1 ... x_n$ and $y_1 ... y_n$ with any duplicates removed. Hence $k < m + n$ expresses that there is at least one variable from $x_1 ... x_n$ equal to one from $y_1 ... y_m$. This condition (that an argument of the term matches an argument of the pattern) is a heuristic which excludes unhelpful rewrites.

Recall that this antecedent truncation fusion rewrite requires that $\mathcal{C}$ be strict. This is always the case in the above rule as $\mathcal{C}[\bot]$ is a pattern match that returns $\bot$ for every branch, which is equivalent to $\bot$.

Rule 31 (sink assert).

$$
\Gamma, \Phi, \mathcal{H} \vdash \begin{pmatrix} \text{assert } p \leftarrow M \text{ in} \\ \quad \text{case } M' \text{ of} \\ \qquad p_1 \rightarrow A_1\ ... \\ \qquad p_n \rightarrow A_n \end{pmatrix} \xrightarrow{\mathfrak{R}} \begin{pmatrix} \text{case } M' \text{ of} \\ \quad p_1 \rightarrow \text{assert } p \leftarrow M \text{ in } A_1\ ... \\ \quad p_n \rightarrow \text{assert } p \leftarrow M \text{ in } A_n \end{pmatrix}
$$

This rule pushes assertions into the branches of pattern matches. It is used by fact fusion to push the fact it is fusing into the fixed-point body it is fusing it into. In general we cannot commute pattern matches like this while preserving $\sqsubseteq$ or $\sqsupseteq$, but since the outer match is an assertion this step preserves equivalence.

Example 5.10. This is a truncation fusion rewrite which has been invoked within a fact fusion rewrite. In this example we are fusing the fact that $\texttt{False} \sqsubseteq \texttt{lq}\, x\, y$ into $\texttt{lq}\, y\, x$.

$$
\begin{array}{l}
\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{lq}'\, \texttt{lq}^a\, y\, x \\
\qquad \xrightarrow{\sqsubseteq}_+ \{ \text{ by definition of } \texttt{lq}' \text{ and beta reduction } \} \\
\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in} \\
\quad \text{case } y, x \text{ of} \\
\qquad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \texttt{True} \\
\texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\
\texttt{Suc}\, y', \texttt{0} & \rightarrow & \texttt{False} \\
\texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & \texttt{lq}^a\, y'\, x'
\end{array} \\
\qquad \xrightarrow{\sqsubseteq}_+ \{ \text{ by sink assert } \} \\
\text{fn } x, y.\ \text{case } y, x \text{ of} \\
\quad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{True} \\
\texttt{0}, \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{True} \\
\texttt{Suc}\, y', \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{False} \\
\texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{lq}^a\, y'\, x'
\end{array} \\
\qquad \xrightarrow{\sqsubseteq}_+ \{ \text{ by case-var substitution } \} \\
\text{fn } x, y.\ \text{case } y, x \text{ of} \\
\quad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, \texttt{0}\, \texttt{0} \text{ in } \texttt{True} \\
\texttt{0}, \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, (\texttt{Suc}\, x')\, \texttt{0} \text{ in } \texttt{True} \\
\texttt{Suc}\, y', \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, \texttt{0}\, (\texttt{Suc}\, y') \text{ in } \texttt{False} \\
\texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, (\texttt{Suc}\, x')\, (\texttt{Suc}\, y') \text{ in } \texttt{lq}^a\, y'\, x'
\end{array} \\
\qquad \xrightarrow{\sqsubseteq}_+ \{ \text{ by unroll fix } \} \\
\text{fn } x, y.\ \text{case } y, x \text{ of} \\
\quad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{True} \text{ in } \texttt{True} \\
\texttt{0}, \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{False} \text{ in } \texttt{True} \\
\texttt{Suc}\, y', \texttt{0} & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{True} \text{ in } \texttt{False} \\
\texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x'\, y' \text{ in } \texttt{lq}^a\, y'\, x'
\end{array} \\
\qquad \xrightarrow{\sqsubseteq}_+ \{ \text{ by case-con reduction } \} \\
\text{fn } x, y.\ \text{case } y, x \text{ of} \\
\quad \begin{array}{lcl}
\texttt{0}, \texttt{0} & \rightarrow & \bot \\
\texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\
\texttt{Suc}\, y', \texttt{0} & \rightarrow & \bot \\
\texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x'\, y' \text{ in } \texttt{lq}^a\, y'\, x'
\end{array}
\end{array}
$$

$$\xrightarrow{\sqsubseteq}_{+} \{ \text{ by folding } (\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \texttt{lq}^{a}\, y\, x) \sqsubseteq h \ \}$$

$$\begin{array}{l} \text{fn } x, y.\ \text{case } y, x \text{ of} \\ \quad \begin{array}{lll} \texttt{0}, \texttt{0} & \rightarrow & \bot \\ \texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\ \texttt{Suc}\, y', \texttt{0} & \rightarrow & \bot \\ \texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & h\, x'\, y' \end{array} \end{array}$$

$$(\text{fn } x, y.\ \text{assert } \texttt{False} \leftarrow \texttt{lq}\, x\, y \text{ in } \Box\, y\, x) \oplus \texttt{lq}$$

$$\xrightarrow{\sqsubseteq}$$

$$\text{fix} \left(\begin{array}{l} \text{fn } h, x, y.\ \text{case } y, x \text{ of} \\ \quad \begin{array}{lll} \texttt{0}, \texttt{0} & \rightarrow & \bot \\ \texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\ \texttt{Suc}\, y', \texttt{0} & \rightarrow & \bot \\ \texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & h\, x'\, y' \end{array} \end{array} \right)$$

Having just shown a fact fusion step which fuses $\texttt{False} \sqsubseteq \texttt{lq}\, x\, y$ into $\texttt{lq}\, y\, x$, here is a context in which this step could have occurred.

$$\texttt{or}\ (\texttt{lq}\, x\, y)\ (\texttt{lq}\, y\, x)$$

$$= \{ \text{ by definition of } \texttt{or} \ \}$$

$$\text{if } \texttt{lq}\, x\, y \text{ then } \texttt{True} \text{ else } \texttt{lq}\, y\, x$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by the fact fusion rewrite from above } \}$$

$$\text{if } \texttt{lq}\, x\, y \text{ then } \texttt{True} \text{ else fix} \left(\begin{array}{l} \text{fn } h, x, y.\ \text{case } y, x \text{ of} \\ \quad \begin{array}{lll} \texttt{0}, \texttt{0} & \rightarrow & \bot \\ \texttt{0}, \texttt{Suc}\, x' & \rightarrow & \texttt{True} \\ \texttt{Suc}\, y', \texttt{0} & \rightarrow & \bot \\ \texttt{Suc}\, y', \texttt{Suc}\, x' & \rightarrow & h\, x'\, y' \end{array} \end{array} \right) y\, x$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by subterm fission (given later on page 90) } \}$$

$$\text{if } \texttt{lq}\, x\, y \text{ then } \texttt{True} \text{ else } \texttt{True}$$

Example 5.11. This example shows another truncated fusion step invoked by a fact fusion step. In this case we are fusing that $\texttt{False} \sqsubseteq \texttt{elem}_n\, xs$ into $\texttt{elem-snoc}_{n,y}\, xs$, which is to say fusing that n is not an element of xs into a function which checks if n is an element of the list xs with y appended to the end. $\texttt{elem-snoc}_{n,y}\, xs$ is equivalent to $\texttt{elem}_n\ (\texttt{snoc}_y\, xs)$; the former is the result of running fixed-point fusion on the latter. All term definitions are given in Appendix A.

```
fn $xs$. assert False ← elem$_n$ $xs$ in
  elem-snoc$'_{n,y}$ elem-snoc$^a_{n,y}$ $xs$

    $\xrightarrow{\sqsubseteq}$ { by definition of elem-snoc$'_{n,y}$ and beta reduction }

fn $xs$. assert False ← elem$_n$ $xs$ in
  case $xs$ of
    [ ]        →  eq $n$ $y$
    $x$ :: $xs'$  →  or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}$ { by sink assert }

fn $xs$. case $xs$ of
  [ ]        →  assert False ← elem$_n$ $xs$ in eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← elem$_n$ $xs$ in
                   or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}$ { by case-var substitution }

fn $xs$. case $xs$ of
  [ ]        →  assert False ← elem$_n$ [ ] in eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← elem$_n$ ($x$ :: $xs'$) in
                   or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}_+$ { by unroll fix }

fn $xs$. case $xs$ of
  [ ]        →  assert False ← False in eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← or (eq $n$ $x$) (elem$_n$ $xs'$) in
                   or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}$ { by case-con reduction }

fn $xs$. case $xs$ of
  [ ]        →  eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← or (eq $n$ $x$) (elem$_n$ $xs'$) in
                   or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}$ { by float case-case }

fn $xs$. case $xs$ of
  [ ]        →  eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← eq $n$ $x$ in
                 assert False ← elem$_n$ $xs'$ in
                   or (eq $n$ $x$) (elem-snoc$^a_{n,y}$ $xs'$)

    $\xrightarrow{\sqsubseteq}$ { by rewrite pattern }

fn $xs$. case $xs$ of
  [ ]        →  eq $n$ $y$
  $x$ :: $xs'$  →  assert False ← eq $n$ $x$ in
                 assert False ← elem$_n$ $xs'$ in
                   elem-snoc$^a_{n,y}$ $xs'$
```

$$\xrightarrow{\sqsubseteq} \left\{\begin{array}{l}\text{by folding}\\ (\text{fn } xs.\ \text{assert } \texttt{False} \leftarrow \texttt{elem}_n\ xs \text{ in } \texttt{elem-snoc}^a_{n,y}\ xs) \sqsubseteq h\end{array}\right\}$$

$$\begin{array}{l}\text{fn } xs.\ \text{case } xs \text{ of}\\ \quad \begin{array}{lcl}\texttt{[]} & \to & \texttt{eq}\, n\, y\\ x :: xs' & \to & \text{assert } \texttt{False} \leftarrow \texttt{eq}\, n\, x \text{ in } h\, xs'\end{array}\end{array}$$

$$\overline{\text{fn } xs.\ \text{assert } \texttt{False} \leftarrow \texttt{elem}_n\ xs \text{ in } \Box\, xs \oplus \texttt{elem-snoc}_{n,y}}$$

$$\xrightarrow{\sqsubseteq}$$

$$\text{fix}\left(\begin{array}{l}\text{fn } h, xs.\ \text{case } xs \text{ of}\\ \quad \begin{array}{lcl}\texttt{[]} & \to & \texttt{eq}\, n\, y\\ x :: xs' & \to & \text{assert } \texttt{False} \leftarrow \texttt{eq}\, n\, x \text{ in } h\, xs'\end{array}\end{array}\right)$$

Having just showed a fact fusion step which fused the fact that n was not an element of xs into $\texttt{elem-snoc}_{n,y}\ xs$, here is a context in which this step could occur.

$$\texttt{or}\,(\texttt{elem}_n\ xs)\,(\texttt{elem}_n\ (\texttt{snoc}_y\ xs))$$

$$= \{ \text{ by definition of } \texttt{or} \}$$

$$\text{if } \texttt{elem}_n\ xs \text{ then } \texttt{True} \text{ else } \texttt{elem}_n\ (\texttt{snoc}_y\ xs)$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by fix-fix fusion } \}$$

$$\text{if } \texttt{elem}_n\ xs \text{ then } \texttt{True} \text{ else } \texttt{elem-snoc}_{n,y}\ xs$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by fact fusion in Example 5.11 } \}$$

$$\begin{array}{l}\text{if } \texttt{elem}_n\ xs \text{ then } \texttt{True} \text{ else}\\ \quad \text{fix}\left(\begin{array}{l}\text{fn } h, xs.\ \text{case } xs \text{ of}\\ \quad \begin{array}{lcl}\texttt{[]} & \to & \texttt{eq}\, n\, y\\ x :: xs' & \to & \text{assert } \texttt{False} \leftarrow \texttt{eq}\, n\, x \text{ in } h\, xs'\end{array}\end{array}\right)\ xs\end{array}$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by subterm fission (given later on page 90) } \}$$

$$\text{if } \texttt{elem}_n\ xs \text{ then } \texttt{True} \text{ else } \texttt{eq}\, n\, y$$

Chapter 6

Fission

The previous chapter introduced fusion, a set of rules which merge contexts into fixed-points with the aim of producing fixed-point promoted form. These fusion steps all add a fact to our fact environment, which is applied using the folding rule. If this fact is never applied in our fusion rewrite then fusion fails. Since the goal of our rewrites is to produce FPPF, and since fusion is the process by which we do this, being able to apply facts with folding is very important.

We can think of fusion as making a fixed-point more complex by fusing more behaviour into it. Here we give rules of the opposite shape, those which simplify fixed-points by fissioning a context out of them. While this seems to be moving away from our goal of fixed-point promoted form the first two fission rules remove fixed-points entirely, and while FPPF is good for our proof process, no fixed-point is even better. The latter two fission rules facilitate the application of folding, and hence facilitate the fusion process. As you might have noticed, many of the examples in the previous chapter had a fission rewrite immediately preceding a folding rewrite.

Fission rules all have the general shape

$$\frac{H\,\mathcal{C}[g] \xrightarrow{\mathfrak{R}}_{+} \mathcal{C}[G]}{\mathrm{fix}^{a}\,(H) \xrightarrow{\mathfrak{R}} \mathcal{C}[\mathrm{fix}^{a}\,(\mathrm{fn}\ g.\, G)]}$$

Starting with a fixed-point $\mathrm{fix}^{a}\,(H)$, each fission rule guesses a potentially fissionable context $\mathcal{C}$. It then rewrites $H\,\mathcal{C}[g]$ and, if the result of this has the shape $\mathcal{C}[G]$ for some G, we know that we can rewrite $\mathrm{fix}^{a}\,(H)$ to $\mathcal{C}[\mathrm{fix}^{a}\,(\mathrm{fn}\ g.\, G)]$. Starting with H and $\mathcal{C}$, fission discovers G by rewriting $H\,\mathcal{C}[g]$.

The first two rules fission out gapless contexts (terms), which has the effect of removing a fixed-point altogether. Picking $\mathcal{C}$ to be a term A, and setting $\mathfrak{R} = \sqsubseteq$ for soundness, we get the shape of these first two rules.

$$\frac{H\ A \xrightarrow{\sqsubseteq}_{+} A}{\mathrm{fix}^{a}\,(H) \xrightarrow{\sqsubseteq} A}$$

These fission rules are described in terms of truncated fixed-points, but all of the rules in this chapter are also defined for untruncated (least) fixed-points. To generate the equivalent fission rule for least fixed-points simply remove all truncation meta-variables from the definition. Below I outline my four fission rules, the first three of which are only sound for $\xrightarrow{\sqsubseteq}$ but the last one holds for $\sqsupseteq$ also.

(Section 6.1) Identity fission removes a fixed-point, replacing it with a non-recursive identity function. In this case the fissioned context $\mathcal{C}$ is gapless, so there is no $\text{fix}^a\,(G)$ term. In rewriting $\texttt{rev}\ (\texttt{rev}\ xs) \xrightarrow{\sqsubseteq}_+ xs$, fix-fix fusion merges the two `rev` functions into one fixed-point, and identity fission rewrites this fixed-point to the identity function

(Section 6.2) Subterm fission also removes fixed-points, in this case by rewriting them to one of their own subterms. In rewriting $\texttt{eq}\,x\,x \xrightarrow{\sqsubseteq}_+ \texttt{True}$, repeated-argument fusion pushes the repetition of x into the body of `eq`, resulting in a fixed-point which only returns `True`; subterm fission removes this recursion, returning just `True`.

(Section 6.3) Constructor fission identifies when a fixed-point will always return the same constructor term and fissions out this constructor. For example, $\texttt{rev}\ (\texttt{snoc}_y\ xs) \xrightarrow{\sqsubseteq}_+ y\ \texttt{::}\ \texttt{rev}\,xs$ uses constructor fission to bring $y\ \texttt{::}\ \square$ topmost, inventing the definition of `rev` in the process.

(Section 6.4) Accumulation fission is used to extract contexts from accumulating parameters. In the rewrite $\texttt{map}_f\ (\texttt{it-rev}\ xs\ ys) \xrightarrow{\sqsupseteq}_+ \texttt{it-rev-map}_f\ xs\ (\texttt{map}_f\ ys)$, fix-fix fusion merges the $\texttt{map}_f$ function into `it-rev`, and accumulation fission pulls it back out of the accumulating argument. It is also pivotal in rewriting $\texttt{it-rev}\,xs\ \texttt{[ ]} \xrightarrow{\mathfrak{R}}_+ \texttt{rev}\,xs$, but this example also requires another technique detailed later in Chapter 8.

6.1 Identity fission

If a fixed-point $\text{fix}^a\,(H)$ always returns something $\sqsubseteq$ to its ith argument, regardless of its other inputs, then we can use identity fission to replace $\text{fix}^a\,(H)$ with $\text{fn}\ x_1, ...x_n.\ x_i$. This step is only sound for $\xrightarrow{\sqsubseteq}$ and not $\xrightarrow{\sqsupseteq}$.

Rule 32 (identity fission).

$$\frac{\Gamma, \Phi, (\mathcal{H}, \text{fix}^a\,(H)) \vdash H\ (\text{fn}\ x_1, ..., x_n.\ x_i) \xrightarrow{\sqsubseteq}_+ \text{fn}\ x_1, ..., x_n.\ x_i}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^a\,(H) \xrightarrow{\sqsubseteq} \text{fn}\ x_1, ..., x_n.\ x_i}$$

$$\begin{array}{ll} \text{if} & x_1...x_n \text{ are fresh variables} \\ & \Gamma \vdash \text{fix}^a\,(H) : \tau_1 \to ... \to \tau_n \to \tau_i \end{array}$$

Example 6.1. Using identity fission to rewrite $\texttt{add}\,x\ \texttt{0}$ to x.

$$\left(\begin{array}{l} \text{fn}\ h, x.\ \text{case}\ x\ \text{of} \\ \quad \texttt{0} \to \texttt{0} \\ \quad \texttt{Suc}\,x' \to \texttt{Suc}\ (h\ x') \end{array}\right)\ (\text{fn}\ x.\ x)$$

$\xrightarrow{\sqsubseteq}$ { by beta reduction }

fn x. case x of
  $\mathtt{0} \to \mathtt{0}$
  $\mathtt{Suc}\, x' \to \mathtt{Suc}\, ((\text{fn } x.\, x)\, x')$

$\xrightarrow{\sqsubseteq}$ { by beta reduction }

fn x. case x of
  $\mathtt{0} \to \mathtt{0}$
  $\mathtt{Suc}\, x' \to \mathtt{Suc}\, x'$

$\xrightarrow{\sqsubseteq}$ { by identity case }

fn x. x

$$\frac{}{(\mathtt{add_0} =_\alpha) \quad \text{fix} \begin{pmatrix} \text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \mathtt{0} \to \mathtt{0} \\ \quad \mathtt{Suc}\, x' \to \mathtt{Suc}\, (h\, x') \end{pmatrix} \xrightarrow{\sqsubseteq} \text{fn } x.\, x}$$

Using constant-argument fusion we can rewrite

$$\mathtt{add}\, x\, \mathtt{0} \xrightarrow{\sqsubseteq} \mathtt{add_0}\, x$$

Combining this with the above identity fission step gives us

$$\mathtt{add}\, x\, \mathtt{0} \xrightarrow{\sqsubseteq}_{+} x$$

6.2 Subterm fission

The rewrite subterm fission attempts to detect whether a fixed-point always returns the same term, regardless of its inputs. It guesses this potential term by calling guessSubterm(fix (H)), detailed in Definition 6.2, and which depends upon the explore function given in Definition 6.1.

Definition 6.1 (Exploring the return values of a fixed-point body).

$$\mathsf{explore}_f\left(\text{case } M \text{ of } P_1 \to B_1 \ldots P_n \to B_n\right) = \bigcup_{i \leq n} \mathtt{explore}_f(P_i)$$
$$\mathsf{explore}_f(A) = \begin{cases} \emptyset & \text{if } f \in \mathsf{freeVars}\,(A) \lor A = \bot \\ \{A\} & \text{otherwise} \end{cases}$$

The meta-level $\mathsf{explore}_f$ function moves into the branches of a fixed-point body term and gives the set of all possible return values which do not depend upon a recursive call to the fixed-point, where this recursive call is given by f. This function is used in the guessSubterm function within this section, and the guessConstructor function which is given later.

Definition 6.2 (Guessing a sub-term into which to rewrite a fixed-point).

$$\begin{aligned}&\mathsf{guessSubterm}\,(\text{fix}\,(\text{fn}\ f, x_1, ..., x_n.\ A)) = \text{fn}\ x_1, ..., x_n.\ B\\ &\qquad \text{if}\ \{B\} = \mathsf{explore}_f(A)\end{aligned}$$

The guessSubterm function conjectures a sub-term which the fixed-point will always return given any argument. It does this by using the explore function to enumerate the set of potential return values, and sees if this set contains only one element. This function does not have to be proven to obey any properties, as the soundness of the subterm fission rewrite in which it is used does not depend upon any properties of this function. If this function guesses an invalid term then subterm fission will simply fail.

Example 6.2. In the term below (from Example 5.10 on page 83) every return value is either `True` or $\perp$, so the guessSubterm operator conjectures `True` as the subterm this fixed-point will always return.

$$\text{fn}\ x, y.\ \texttt{True} \quad = \quad \mathsf{guessSubterm}\left(\text{fix}\left(\begin{array}{lll}\multicolumn{3}{l}{\text{fn}\ f, y, x.\ \text{case}\ y, x\ \text{of}}\\ \quad \mathtt{0}, \mathtt{0} & \to & \perp\\ \quad \mathtt{0}, \mathtt{Suc}\ x' & \to & \texttt{True}\\ \quad \mathtt{Suc}\ y', \mathtt{0} & \to & \perp\\ \quad \mathtt{Suc}\ y', \mathtt{Suc}\ x' & \to & f\ y'\ x'\end{array}\right)\right)$$

Example 6.3. Here is a tagged fixed-point (from Example 5.11 on page 84) in which the subterm to be returned is a not a constructor:

$$\text{fn}\ xs.\ \mathsf{eq}\ x\ y \quad = \quad \mathsf{guessSubterm}\left(\text{fix}\left(\begin{array}{l}\text{fn}\ h, xs.\ \text{case}\ xs\ \text{of}\\ \quad \mathtt{[\,]} \to \mathsf{eq}\,n\,y\\ \quad x :: xs' \to\\ \qquad \text{assert}\ \texttt{False} \leftarrow \mathsf{eq}\,n\,x\ \text{in}\ h\,xs'\end{array}\right)\right)$$

Rule 33 (subterm fission).

$$\frac{\Gamma, \Phi, (\mathcal{H}, \text{fix}^a\,(H)) \vdash H\ A \xrightarrow{\sqsubseteq}_{+} A}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^a\,(H) \xrightarrow{\sqsubseteq} A}$$

$$\text{if}\quad A = \mathsf{guessSubterm}\,(\text{fix}^a\,(H))$$

Elea conjectures that $\text{fix}^a\,(H)$ can be rewritten to subterm A, and uses its own rewriting algorithm to check if this rewrite is valid. This does not require any correctness properties to be proven of guessSubterm, if guessSubterm returns an invalid term then the antecedent rewrite $H\ A \xrightarrow{\sqsubseteq}_{+} A$ will fail.

Example 6.4. This example uses subterm fission on the result of the fact fusion step on page 84, rewriting a fixed-point which always returns `True` into just `True`.

$$\left(\begin{array}{llll}\text{fn } f, y, x.\ \text{case } y, x \text{ of} & & & \\ \quad \mathtt{0}, \mathtt{0} & \to & \bot & \\ \quad \mathtt{0}, \mathtt{Suc}\ x' & \to & \mathtt{True} & \\ \quad \mathtt{Suc}\ y', \mathtt{0} & \to & \bot & \\ \quad \mathtt{Suc}\ y', \mathtt{Suc}\ x' & \to & f\ y'\ x' & \end{array}\right) (\text{fn } x, y.\ \mathtt{True})$$

$\xrightarrow{\sqsubseteq}_+$ { by beta reduction }

$$\begin{array}{lll}\text{fn } y, x.\ \text{case } y, x \text{ of} & & \\ \quad \mathtt{0}, \mathtt{0} & \to & \bot \\ \quad \mathtt{0}, \mathtt{Suc}\ x' & \to & \mathtt{True} \\ \quad \mathtt{Suc}\ y', \mathtt{0} & \to & \bot \\ \quad \mathtt{Suc}\ y', \mathtt{Suc}\ x' & \to & \mathtt{True} \end{array}$$

$\xrightarrow{\sqsubseteq}$ { by constant case }

$$\begin{array}{l}\text{fn } x, y.\ \text{case } y \text{ of} \\ \quad \mathtt{0} \to \bot \\ \quad \mathtt{Suc}\ y' \to \mathtt{True} \end{array}$$

$\xrightarrow{\sqsubseteq}$ { by constant case }

fn x, y. `True`

$$\dfrac{}{\text{fix}\left(\begin{array}{lll}\text{fn } f, y, x.\ \text{case } y, x \text{ of} & & \\ \quad \mathtt{0}, \mathtt{0} & \to & \bot \\ \quad \mathtt{0}, \mathtt{Suc}\ x' & \to & \mathtt{True} \\ \quad \mathtt{Suc}\ y', \mathtt{0} & \to & \bot \\ \quad \mathtt{Suc}\ y', \mathtt{Suc}\ x' & \to & f\ y'\ x' \end{array}\right) \xrightarrow{\sqsubseteq} \text{fn } x, y.\ \mathtt{True}}$$

6.3 Constructor fission

Constructor fission attempts to guess if a fixed-point always returns the same constructor regardless of its inputs and will try to fission this constructor to be topmost in the term. For example, the fixed-point below always returns the successor of some term.

$$\text{fix}\left(\begin{array}{l}\text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \mathtt{0} \to \mathtt{Suc}\ y \\ \quad \mathtt{Suc}\ x' \to \mathtt{Suc}\ (h\ x') \end{array}\right) \sqsubseteq \text{fn } x.\ \mathtt{Suc}\ (...)$$

Using constructor fission we can pull this $\texttt{Suc}\ (...)$ to the top of the term.

$$\text{fix}\begin{pmatrix}\text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (h\ x')\end{pmatrix}$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by constructor fission } \}$$

$$\text{fn } x.\ \texttt{Suc}\left(\text{fix}\begin{pmatrix}\text{fn } g, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (g\ x')\end{pmatrix} x\right)$$

Given a fixed-point $\text{fix}^a\ (H)$, constructor fission calls $\mathsf{guessConstructor}(\text{fix}^a\ (H))$, detailed in Definition 6.3, to conjecture the constructor context which can be fissioned out. As well as the rule constructor fission, I also define two more auxiliary rewrites, float context and beta-abstract context. These rewrite rules are restricted to only occur at the very end of the antecedent rewrite to constructor fission. They attempt to bring the context we are fissioning out to be topmost in the term so that constructor fission succeeds. This section finishes with two examples.

Definition 6.3 (Guessing a constructor to fission out of a fixed-point).

$$\begin{aligned}&\mathsf{guessConstructor}\,(\text{fix}\,(\text{fn } f, x_1, ..., x_n.\ A)) = \\ &\qquad\qquad \text{fn } x_1, ..., x_n.\ .\ \text{con}_j\langle \boldsymbol{T}\rangle\ B_1 ... B_{i-1}\ \square\ B_i ... B_n \\ &\quad \text{if} \quad A : \boldsymbol{T} \\ &\qquad\quad \forall (C \in \mathsf{explore}_f(A))\ .\ \exists D\ .\ C = \text{con}_j\langle \boldsymbol{T}\rangle\ B_1 ... B_{i-1}\ D\ B_i ... B_n\end{aligned}$$

The function $\mathsf{guessConstructor}$ uses the $\mathsf{explore}$ function (Definition 6.1) to enumerate a subset of the return values of its argument. If all of these potential return values $C \in \mathsf{explore}_f(A)$ are the same constructor $\text{con}_j\langle \boldsymbol{T}\rangle$, and all have the same arguments except for one argument position i, then return the context where the gap is this argument position. In Elea's implementation of this function the terms $B_1 ... B_n$, and the value of j, are chosen by pulling a random element from $\mathsf{explore}_f(A)$ and inspecting its shape.

Example 6.5. Examples of the $\mathsf{guessConstructor}$ operator.

$$\text{fn } x.\ \texttt{Suc}\ (\square\ x) \quad = \quad \mathsf{guessConstructor}\left(\text{fix}\begin{pmatrix}\text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (h\ x')\end{pmatrix}\right)$$

$$\text{fn } xs.\ y :: \square\ xs \quad = \quad \mathsf{guessConstructor}\left(\text{fix}\begin{pmatrix}\text{fn } h, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \texttt{snoc}_x\ (h\ xs')\end{pmatrix}\right)$$

Rule 34 (constructor fission).

$$\frac{\Gamma[g \mapsto \tau], \Phi, (\mathcal{H}, \text{fix}^a\,(H)) \vdash H\,\mathcal{C}[g] \xrightarrow{\sqsubseteq}_+ \mathcal{C}[G]}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}^a\,(H) \xrightarrow{\sqsubseteq} \mathcal{C}[\text{fix}^a\,(\text{fn } g.\, G)]}$$

$$\begin{aligned}\text{if}\quad & g \text{ is a fresh variable}\\ & \mathcal{C} = \mathsf{guessConstructor}(\text{fix}^a\,(H))\\ & \tau \text{ is the type of the gap in } \mathcal{C}\\ & H \not\trianglelefteq \text{fn } g.\, G\end{aligned}$$

This rewrite rule fissions a constructor context out of a fixed-point. The not-embedded check $H \not\trianglelefteq \text{fn } g.\, G$ uses the homeomorphic embedding (Definition 2.47 on page 41) to ensure that this step has simplified the fixed-point in some way. For example, it stops us begin able to fission the context $\mathtt{0} :: \Box$ out of $\text{fix}\,(\text{fn } xs.\, \mathtt{0} :: xs)$.

The soundness of this rule not require any correctness properties to be proven of $\mathsf{guessConstructor}$, since if $\mathsf{guessConstructor}$ returns an invalid context then the antecedent rewrite $H\,\mathcal{C}[g] \xrightarrow{\sqsubseteq}_+ \mathcal{C}[G]$ will fail.

6.3.1 Auxiliary rewrites for constructor fission

The antecedent rewrite in the definition of constructor fission is aiming to produce a term of the shape $\mathcal{C}[G]$, where G is any term. The two rules described below are used within constructor fission to help pull this context $\mathcal{C}$ to be topmost, coercing the result into the shape $\mathcal{C}[G]$. They are special rules in that they are only applicable at the end of an antecedent rewrite within constructor fission. The constructor fission examples at the end of this section highlight their usefulness.

Rule 35 (float context).

$$\Gamma, \Phi, \mathcal{H} \vdash \begin{pmatrix}\text{case } M \text{ of} \\ \quad p_1 \to \mathcal{C}[A_1]\ ... \\ \quad p_n \to \mathcal{C}[A_n]\end{pmatrix} \xrightarrow{\sqsubseteq} \mathcal{C}\begin{bmatrix}\text{case } M \text{ of} \\ \quad p_1 \to A_1\ ... \\ \quad p_n \to A_n\end{bmatrix}$$

This rewrite rule will only occur at the end of the antecedent rewrite to constructor fission, where $\mathcal{C}$ is the constructor context we are fissioning.

Rule 36 (beta-abstract context).

$$\Gamma, \Phi, \mathcal{H} \vdash \text{fn } x_1, ..., x_n.\, \mathcal{C}[G] \xrightarrow{\sqsubseteq} \text{fn } x_1, ..., x_n.\, \mathcal{C}[(\text{fn } x_1, ..., x_n.\, G)\; x_1 ... x_n]$$

This rewrite rule will only occur at the end of the antecedent rewrite to constructor fission, where $\text{fn } x_1, ..., x_n.\, \mathcal{C}[\Box\; x_1 ... x_n]$ is the constructor context we are fissioning. This rule preserves denotational equivalence but is only defined for $\sqsubseteq$ since so is constructor fission.

Now that we have given our constructor fission rule, and its two auxiliary rewrite rules, we give two examples of constructor fission in action:

Example 6.6. The constructor fission example from the beginning of this section. The context here is $\mathcal{C} = \text{fn } x.\, \texttt{Suc}\ (\Box\, x)$. This step also appears in the earlier Example 5.2 on page 70.

$$\begin{pmatrix}\text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (h\ x')\end{pmatrix} (\text{fn } x.\, \texttt{Suc}\ (g\ x))$$

$\xrightarrow{\sqsubseteq}_+$ { by beta reduction }

$$\begin{array}{l}\text{fn } x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (\texttt{Suc}\ (g\ x'))\end{array}$$

$\xrightarrow{\sqsubseteq}$ { by float context }

$$\text{fn } x.\, \texttt{Suc} \begin{pmatrix}\text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (g\ x')\end{pmatrix}$$

$\xrightarrow{\sqsubseteq}$ { by beta-abstract context }

$$\text{fn } x.\, \texttt{Suc} \left(\begin{pmatrix}\text{fn } x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (g\ x')\end{pmatrix} x\right)$$

$$\overline{\text{fix} \begin{pmatrix}\text{fn } h, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{Suc}\ y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (h\ x')\end{pmatrix} \xrightarrow{\sqsubseteq} \text{fn } x.\, \texttt{Suc} \left(\text{fix} \begin{pmatrix}\text{fn } g, x.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\ x' \to \texttt{Suc}\ (g\ x')\end{pmatrix} x\right)}$$

Example 6.7. The constructor fission step from the earlier Example 5.4 on page 73. The context here is $\mathcal{C} = \text{fn } xs.\, y :: \Box\, xs$.

$$\begin{pmatrix}\text{fn } h, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \texttt{snoc}_x(h\ xs')\end{pmatrix} (\text{fn } xs.\, y :: g\ xs)$$

$\xrightarrow{\sqsubseteq}_+$ { by beta reduction }

$$\begin{array}{l}\text{fn } xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \texttt{snoc}_x\ (y :: g\ xs')\end{array}$$

$\xrightarrow{\sqsubseteq}$ { by unfold fix }

$$\begin{array}{l}\text{fn } xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to y :: \texttt{snoc}_x\ (g\ xs')\end{array}$$

$$\xrightarrow{\sqsubseteq} \{\text{ by float context }\}$$

$$\text{fn } xs.\, y :: \begin{pmatrix}\text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[ ]} \\ \quad x :: xs' \to \texttt{snoc}_x\,(g\, xs')\end{pmatrix}$$

$$\xrightarrow{\sqsubseteq} \{\text{ by beta-abstract context }\}$$

$$\text{fn } xs.\, y :: \left(\text{fn } xs.\ \begin{pmatrix}\text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[ ]} \\ \quad x :: xs' \to \texttt{snoc}_x\,(g\, xs')\end{pmatrix} xs\right)$$

$$\frac{}{\text{fix}\begin{pmatrix}\text{fn } h, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{[}y\texttt{]} \\ \quad x :: xs' \to \\ \qquad \texttt{snoc}_x\,(h\, xs')\end{pmatrix} \xrightarrow{\sqsubseteq} \text{fn } xs.\, y :: \left(\text{fix}\begin{pmatrix}\text{fn } g, xs.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to y \\ \quad x :: xs' \to \\ \qquad \texttt{snoc}_x\,(g\, xs')\end{pmatrix} xs\right)}$$

6.4 Accumulation fission

An accumulating argument to a fixed-point is one which does not decrease or stay constant in at least one recursive call. The function given in Definition 5.3 returns the argument positions to a fixed-point which are accumulating. Examples are the second arguments of `it-rev` and `it-add` (defined in Appendix A).

Sometimes it will be the case that we can fission out a context applied to an accumulating argument. For example the accumulation fission step below fissions the $\texttt{map}_f\ \Box$ context out of the second argument of the fixed-point:

$$\text{fn } xs, ys.\ \texttt{map}_f\ (\texttt{it-rev}\ xs\ ys)$$

$$\xrightarrow{\sqsupseteq}_+ \{\text{ by fix-fix fusion and eta reduction }\}$$

$$\text{fix}\begin{pmatrix}\text{fn } h, xs, ys.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{map}_f\ ys \\ \quad x :: xs' \to h\, xs'\,(x :: ys)\end{pmatrix}$$

$$\xrightarrow{\sqsupseteq} \{\text{ by accumulation fission }\}$$

$$\text{fn } xs, ys.\ \text{fix}\begin{pmatrix}\text{fn } h, xs, ys.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to h\, xs'\,(f\ x :: ys)\end{pmatrix} xs\ (\texttt{map}_f\ ys)$$

This technique has proven to be experimentally very useful. It allows us to prove various properties about tail recursive functions like `it-rev` and `it-add` (definitions given in Appendix A).

Accumulation fission uses guessAcc to conjecture a context we may be able to fission out of an argument to a fixed-point. It does this by analysing the non-recursive branches of a given fixed-point to see if they return a context applied to an accumulating argument. This function is given in Definition 6.4, along with two examples in Example 6.8.

Definition 6.4 (Guessing a context fissionable out of an accumulating parameter).

$$\begin{array}{l} \mathsf{guessAcc}\,(\mathrm{fix}\,(\mathrm{fn}\ f, x_1, ..., x_n.\ A)) = \mathrm{fn}\ x_1, ..., x_n.\ \Box\ x_1...x_{i-1}\ \mathcal{C}[x_i]\ x_{i+1}...x_n \\ \quad \text{if} \quad \forall(B \in \mathsf{explore}_f(A))\ .\ x_i \in \mathsf{freeVars}\,(A) \Rightarrow B = \mathcal{C}[x_i] \\ \quad\quad \mathcal{C} \text{ is not the identity context} \\ \quad\quad \{x_1, ..., x_{i-1}, x_{i+1}, ..., x_n\} \cap \mathsf{freeVars}\,(\mathcal{C}[x_i]) = \emptyset \end{array}$$

The guessAcc function checks if every potential return value of the given fixed-point is always the same context around a single variable. If this is the case, then we may be able to fission this context out of an accumulating parameter to this function. The implementation of this function within Elea chooses $\mathcal{C}$ and i by iterating through every variable index $1..n$ to see if any match this pattern. Potential return values are found using the explore function from Definition 6.1 on page 89.

Example 6.8.

$$\mathrm{fn}\ xs, ys.\ \Box\ xs\ (\mathtt{map}_f\ ys) \quad = \quad \mathsf{guessAcc}\left(\mathrm{fix}\left(\begin{array}{l}\mathrm{fn}\ h, xs, ys.\ \mathrm{case}\ xs\ \mathrm{of} \\ \quad \mathtt{[\,]} \rightarrow \mathtt{map}_f\ ys \\ \quad x :: xs' \rightarrow h\ xs'\ (x :: ys)\end{array}\right)\right)$$

$$\mathrm{fn}\ x, y.\ \Box\ x\ (\mathtt{double}\ y) \quad = \quad \mathsf{guessAcc}\left(\mathrm{fix}\left(\begin{array}{l}\mathrm{fn}\ h, x, y.\ \mathrm{case}\ x\ \mathrm{of} \\ \quad \mathtt{0} \rightarrow \mathtt{double}\ y \\ \quad \mathtt{Suc}\ x' \rightarrow h\ x'\ (\mathtt{Suc}\ y)\end{array}\right)\right)$$

Rule 37 (Accumulation fission).

$$\frac{\Gamma, \Phi, (\mathcal{H}, \mathrm{fix}^a\,(H)) \vdash H\,\mathcal{C}[g] \xrightarrow{\mathfrak{R}}_+ \mathcal{C}[G]}{\Gamma, \Phi, \mathcal{H} \vdash \mathrm{fix}^a\,(H) \xrightarrow{\mathfrak{R}} \mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\ G)]}$$

$$\begin{array}{l} \text{if} \quad g \text{ is a fresh variable} \\ \quad\quad \mathcal{C} = \mathsf{guessAcc}\,(\mathrm{fix}^a\,(H)) \\ \quad\quad \tau \text{ is the type of the gap in } \mathcal{C} \end{array}$$

Accumulation fission is able to pull a context out of an accumulating argument to a fixed-point. Despite its usefulness this step moves us in the opposite direction to fixed-point promoted form so we define it only for $\xrightarrow{\sqsupseteq}$, even though it is sound for $\xrightarrow{\sqsubseteq}$.

6.4.1 An auxiliary rewrite for accumulation fission

The antecedent rewrite in the definition of accumulation fission is aiming to produce a term of the shape $\mathcal{C}[G]$, where G is any term. The rule described below is used at the end of this antecedent rewrite to help pull this context $\mathcal{C}$ to be topmost. It is a special rule which is only applicable at the end of an antecedent rewrite to accumulation fission.

Rule 38 (beta-abstract accumulation).

$$\begin{array}{l}
\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\sqsupseteq} (\text{fn } x_1, ..., x_n.\, B)\; x_1 \ldots \mathcal{C}'[x_i] \ldots x_n \\
\qquad \text{if} \;\; \mathcal{C} = \text{fn } x_1, ..., x_n.\, \square\, x_1 \ldots \mathcal{C}'[x_i] \ldots x_n \\
\qquad\qquad B[\mathcal{C}'[x_i]/x_i] = A
\end{array}$$

When automating this rule the value of $\mathcal{C}$ is taken from the accumulation fission step it occurs within. Elea discovers B by checking if $[\mathcal{C}'[x_i]/x_i]$ can be applied in reverse to A, meaning that if every occurrence of x_i in A is within $\mathcal{C}'$ then we can replace all occurrences of $\mathcal{C}'[x_i]$ with just x_i within A to get B.

Example 6.9. Using accumulation fission to pull the `double` function out of the accumulating argument of a fixed-point. The context being fissioned here is $\mathcal{C} = \text{fn } x, y.\, \square\, x\, (\texttt{double}\, y)$.

$$\begin{pmatrix} \begin{array}{l} \text{fn } h, x, y.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{double}\, y \\ \quad \texttt{Suc}\, x' \to h\, x'\, (\texttt{Suc}\, y) \end{array} \end{pmatrix} (\text{fn } x, y.\, g\, x\, (\texttt{double}\, y))$$

$$\xrightarrow{\sqsupseteq}_{+} \{ \text{ by beta reduction } \}$$

$$\begin{array}{l} \text{fn } x, y.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{double}\, y \\ \quad \texttt{Suc}\, x' \to g\, x'\, (\texttt{double}\, (\texttt{Suc}\, y)) \end{array}$$

$$\xrightarrow{\sqsupseteq} \{ \text{ by unfold fix } \}$$

$$\begin{array}{l} \text{fn } x, y.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to \texttt{double}\, y \\ \quad \texttt{Suc}\, x' \to g\, x'\, (\texttt{Suc}\, (\texttt{Suc}\, (\texttt{double}\, y))) \end{array}$$

$$\xrightarrow{\sqsupseteq} \{ \text{ by beta-abstract accumulation } \}$$

$$\text{fn } x, y.\ \begin{pmatrix} \begin{array}{l} \text{fn } x, y.\ \text{case } x \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\, x' \to g\, x'\, (\texttt{Suc}\, (\texttt{Suc}\, y)) \end{array} \end{pmatrix} x\, (\texttt{double}\, y)$$

$$\frac{}{\text{fix} \begin{pmatrix} \begin{array}{l} \text{fn } h, x, y. \\ \text{case } x, y \text{ of} \\ \quad \texttt{0} \to \texttt{double}\, y \\ \quad \texttt{Suc}\, x' \to \\ \qquad h\, x'\, (\texttt{Suc}\, y) \end{array} \end{pmatrix} \xrightarrow{\sqsupseteq} \text{fn } x, y.\ \text{fix} \begin{pmatrix} \begin{array}{l} \text{fn } g, x, y. \\ \text{case } x, y \text{ of} \\ \quad \texttt{0} \to y \\ \quad \texttt{Suc}\, x' \to \\ \qquad g\, xs'\, (\texttt{Suc}\, (\texttt{Suc}\, y)) \end{array} \end{pmatrix} x\, (\texttt{double}\, y)}$$

Example 6.10. The accumulation fission step from the beginning of this section. The context being fissioned here is $\mathcal{C} = \text{fn } xs, ys.\, \square\, xs\, (\texttt{map}_f\, ys)$.

$$\begin{pmatrix} \begin{array}{l} \text{fn } h, xs, ys.\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to \texttt{map}_f\, ys \\ \quad x :: xs' \to h\, xs'\, (x :: ys) \end{array} \end{pmatrix} (\text{fn } xs, ys.\, g\, xs\, (\texttt{map}_f\, ys))$$

$\xrightarrow{\sqsupseteq}_{+}$ { by beta reduction }

$$\begin{array}{l}\text{fn } xs, ys.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{map}_f\ ys\\ \quad x :: xs' \to g\ xs'\ (\texttt{map}_f\ (x :: ys))\end{array}$$

$\xrightarrow{\sqsupseteq}$ { by unfold fix }

$$\begin{array}{l}\text{fn } xs, ys.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to \texttt{map}_f\ ys\\ \quad x :: xs' \to g\ xs'\ (f\ x :: \texttt{map}_f\ ys)\end{array}$$

$\xrightarrow{\sqsupseteq}$ { by beta-abstract accumulation }

$$\text{fn } xs, ys.\ \left(\begin{array}{l}\text{fn } xs, ys.\ \text{case } xs \text{ of}\\ \quad \texttt{[ ]} \to ys\\ \quad x :: xs' \to g\ xs'\ (f\ x :: ys)\end{array}\right)\ xs\ (\texttt{map}_f\ ys)$$

$$\text{fix}\left(\begin{array}{l}\text{fn } h, xs, ys.\\ \text{case } xs, ys \text{ of}\\ \quad \texttt{[ ]} \to \texttt{map}_f\ ys\\ \quad x :: xs' \to\\ \qquad h\ xs'\ (x :: ys)\end{array}\right) \xrightarrow{\sqsupseteq} \text{fn } xs, ys.\ \text{fix}\left(\begin{array}{l}\text{fn } g, xs, ys.\\ \text{case } xs, ys \text{ of}\\ \quad \texttt{[ ]} \to ys\\ \quad x :: xs' \to\\ \qquad g\ xs'\ (f\ x :: ys)\end{array}\right)\ xs\ (\texttt{map}_f\ ys)$$

Chapter 7

A rewriting based theorem prover

The overall goal of this thesis is an automated prover for properties of $\sqsubseteq$ between terms. The previous chapters defined a system powerful enough to perform rewrites like `rev` (`rev` xs) $\xrightarrow{\sqsubseteq}_+ xs$ and `add` $x\,x \xrightarrow{\sqsubseteq}_+$ `double` x. Chapter 9 proves that $A \xrightarrow{\sqsubseteq}_+ B$ means $A \sqsubseteq B$, so this rewrite system is a fairly decent $\sqsubseteq$-prover by itself. To prove $A \sqsubseteq B$ we could rewrite $A \xrightarrow{\sqsubseteq}_+ B'$ and check $B =_\alpha B'$.

However, there are a lot of properties just this cannot prove. A re-arrangement of pattern matches or function arguments in the right-hand side of $\sqsubseteq$ for example would remove α-equality, so a property like `eq` $x\,y \sqsubseteq$ `eq` $y\,x$ would not be provable. This chapter describes the theorem prover I have build around my term rewriting system, allowing it to prove properties such as this.

Rather than define a separate property language for νPCF I decided to extend νPCF into a property language over itself, which I call $\nu\text{PCF}^{\sqsubseteq}$. This allows for the re-use a lot of definitions and rewrite rules from the previous chapters, and drastically reduces the amount of new concepts which need to be introduced. Section 7.1 describes the $\nu\text{PCF}^{\sqsubseteq}$ language and then Section 7.2 gives the set of extra rewrite rules defined just for $\nu\text{PCF}^{\sqsubseteq}$. This approach is the same as that of the ACL2 theorem prover [37, 38], which also features a unified term and property language.

7.1 $\nu\text{PCF}^{\sqsubseteq}$

This section details an extension of νPCF into a language of properties over itself, which I call $\nu\text{PCF}^{\sqsubseteq}$, the grammar of which is given in Definition 7.1. Instead of adding a boolean type for properties it uses the single element data-type, where truth is $\bot$ and falsity is the single element (), this is formalised in Definition 7.2. The typing rules and denotation of $\nu\text{PCF}^{\sqsubseteq}$ are an extension of νPCF too (Definition 7.3), but these semantics mean that this language cannot be executed (Lemma 7.1).

Since truth is $\bot$, $\sqsubseteq$ represents backwards implication (Definition 7.4) and hence $\xrightarrow{\sqsubseteq}_+$ is a top-down theorem prover on properties/terms in $\nu\text{PCF}^{\sqsubseteq}$ (Remark 7.1), as it preserves sufficiency. Remark 7.2 discusses why I extended νPCF to $\nu\text{PCF}^{\sqsubseteq}$ instead of creating a separate property language.

Definition 7.1 (νPCF$^{\sqsubseteq}$).

$$
\begin{array}{rcll}
\mathcal{P}, \mathcal{Q}, \mathcal{R} & ::= & A \in \nu\text{PCF} & \nu\text{PCF term} \\
& | & \mathcal{P} \sqsubseteq \mathcal{Q} & \text{approximation} \\
& | & \text{forall } x : \tau .\, \mathcal{P} & \text{universal quantification} \\
& | & \text{case } \mathcal{P} \text{ of } P_1 \to \mathcal{Q}_1 \ldots P_n \to \mathcal{Q}_n & \text{case split}
\end{array}
$$

νPCF$^{\sqsubseteq}$, ranged over by $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$, is a superset of νPCF, inheriting its types, typing rules, and denotational semantics. However, νPCF$^{\sqsubseteq}$ does not, and cannot (Lemma 7.1), have an operational semantics.

Definition 7.2 (Typing νPCF$^{\sqsubseteq}$ properties).

$$
\texttt{Prop} \overset{\text{def}}{=} (()) \qquad \texttt{tt} \overset{\text{def}}{=} \bot_{\texttt{Prop}} \qquad \texttt{ff} \overset{\text{def}}{=} \text{con}_1\langle\texttt{Prop}\rangle
$$

The type of properties in νPCF$^{\sqsubseteq}$, `Prop`, is given as the single element data-type, where truth is undefinedness and falsity by the single element $\text{con}_1\langle\texttt{Prop}\rangle$.

Definition 7.3 (νPCF$^{\sqsubseteq}$ typing and denotation).

$$
\frac{\Gamma \vdash \mathcal{P} : \tau \qquad \Gamma \vdash \mathcal{Q} : \tau}{\Gamma \vdash \mathcal{P} \sqsubseteq \mathcal{Q} : \texttt{Prop}} \qquad \frac{\Gamma[x \mapsto \tau] \vdash \mathcal{P} : \texttt{Prop}}{\Gamma \vdash \text{forall } x : \tau .\, \mathcal{P} : \texttt{Prop}}
$$

$$
[\![\mathcal{P} \sqsubseteq \mathcal{Q}]\!]\,\rho \overset{\text{def}}{=} \begin{cases} \texttt{tt} & \text{if } [\![\mathcal{P}]\!]\,\rho \sqsubseteq [\![\mathcal{Q}]\!]\,\rho \\ \texttt{ff} & \text{otherwise} \end{cases}
$$

$$
[\![\text{forall } x : \tau .\, \mathcal{P}]\!]\,\rho \overset{\text{def}}{=} \begin{cases} \texttt{tt} & \text{if } \forall (d \in [\![\tau]\!]) \,.\, [\![\mathcal{P}]\!]\,\rho[x \mapsto d] = \texttt{tt} \\ \texttt{ff} & \text{otherwise} \end{cases}
$$

The grammar of νPCF$^{\sqsubseteq}$ is an extension of νPCF, adding two constructs, $\sqsubseteq$ and forall. Due to this, I can re-use all the typing rules and denotational semantics of νPCF in νPCF$^{\sqsubseteq}$, giving only the typing and denotation for $\sqsubseteq$ and forall above.

Definition 7.4 (Boolean algebra in νPCF$^{\sqsubseteq}$).

$$
\begin{array}{lcl}
\mathcal{P} \Rightarrow \mathcal{Q} & \overset{\text{def}}{=} & \mathcal{Q} \sqsubseteq \mathcal{P} \\
\neg \mathcal{P} & \overset{\text{def}}{=} & \mathcal{P} \Rightarrow \texttt{ff} \\
\mathcal{P} \vee \mathcal{Q} & \overset{\text{def}}{=} & \text{seq}\, \mathcal{P} \text{ in } \mathcal{Q} \\
\mathcal{P} \wedge \mathcal{Q} & \overset{\text{def}}{=} & \neg(\neg \mathcal{P} \vee \neg \mathcal{Q})
\end{array}
$$

Using $\sqsubseteq$ and seq...in from Definition 2.17 on page 24 I define the usual boolean algebra on terms of the type `Prop`.

Lemma 7.1 (If $\nu\text{PCF}^{\sqsubseteq}$ is denotationally adequate it is not executable). Assume we could evaluate $\nu\text{PCF}^{\sqsubseteq}$ while preserving denotational adequacy (Lemma 2.10 on page 38). We would have

$$[\![\neg(A \sqsubseteq \bot)]\!] = [\![\texttt{ff}]\!] \;\Rightarrow\; \neg(A \sqsubseteq \bot) \Downarrow_{\texttt{Prop}} \texttt{ff}$$

So if $A \not\sqsubseteq \bot$ is false, viz. $A = \bot$, then our program terminates with the value `ff`, giving us a solution to the halting problem. $\square$

Remark 7.1 ($\xrightarrow{\sqsubseteq}_+$ is a top-down theorem prover). A theorem prover is top-down if it internally it transforms goals into sufficient sub-goals, meaning that a proof of the sub-goals is a proof of the original goal. In $\nu\text{PCF}^{\sqsubseteq}$, for any $\mathcal{P}$: `Prop` if we can rewrite $\mathcal{P} \xrightarrow{\sqsubseteq}_+ \mathcal{Q}$ we have $\mathcal{P} \sqsubseteq \mathcal{Q}$ and hence $\mathcal{Q} \Rightarrow \mathcal{P}$ (Definition 7.4), so $\xrightarrow{\sqsubseteq}_+$ rewrites goals into sufficient sub-goals, and so is a top-down theorem prover.

Lemma 7.2 (How Elea works). Given well-typed closed terms A and B in νPCF:

$$\text{if} \quad A \sqsubseteq B \xrightarrow{\sqsubseteq}_+ \texttt{tt} \qquad \text{then} \quad A \precsim B$$

If we can rewrite $A \sqsubseteq B$ to `tt`, we have shown A observationally approximates B.

Proof. From the denotation of $A \sqsubseteq B$ and `tt`, and that $\xrightarrow{\sqsubseteq}_+$ preserves $\sqsubseteq$ (Chapter 9) and therefore $\Leftarrow$ (Definition 7.4) we have that $A \sqsubseteq B \Leftarrow B$, and hence $A \sqsubseteq B$. Then, by Theorem 1 we have $A \precsim B$. $\square$

Remark 7.2 (Why $\nu\text{PCF}^{\sqsubseteq}$ and not a separate property language?). The primary reason for this choice was laziness, but of the human type rather than the language. Most of the earlier rewrite rules are still sound for $\nu\text{PCF}^{\sqsubseteq}$, and are just as useful for theorem proving as they are for term rewriting. It also means $\nu\text{PCF}^{\sqsubseteq}$ can inherit the typing rules and denotational semantics of νPCF.

Brevity aside, one nice feature of this is modelling case-analysis as pattern-matching in $\nu\text{PCF}^{\sqsubseteq}$, in particular that it returns `tt` if the term we are analysing does not terminate. This allows us to float pattern-matches out of the left-hand side of $\sqsubseteq$ properties (Rule 40 on page 102) which synergises excellently with the least pre-fixed-point principle (Rule 46 on page 103).

7.2 Rewriting $\nu\text{PCF}^{\sqsubseteq}$ to prove theorems

Now that the previous section has given the $\nu\text{PCF}^{\sqsubseteq}$ language used to represent properties over νPCF, and explained that $\xrightarrow{\sqsubseteq}_+$ is a top-down theorem prover over this language, this section defines the $\xrightarrow{\sqsubseteq}$ rewrite rules which perform this theorem proving.

As mentioned in the previous section, one of the reasons for extending νPCF into $\nu\text{PCF}^{\sqsubseteq}$ is that we can re-use many of the rewrite rules we have defined over νPCF on $\nu\text{PCF}^{\sqsubseteq}$. The property we have lost in adding $\sqsubseteq$ to νPCF is monotonicity, and hence continuity, of term denotations (Definition 2.27 on page 28), so the re-usable rewrites are those which do not rely on this property. $\nu\text{PCF}^{\sqsubseteq}$ also does not need rules which refer to constructs only existing in νPCF, namely fixed-points and term application.

With this is mind, the rules from νPCF which Elea re-uses for $\nu\text{PCF}^{\sqsubseteq}$ are as follows: case reduction, float case-case, constant case, undefined, traverse lambda, traverse var-branch, traverse branch and apply pattern.

The rest of this section defines extra rewrite rules which apply only to $\nu\text{PCF}^{\sqsubseteq}$ superset, the theorem proving rewrites.

Rule 39 (bottom).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \bot \sqsubseteq A \xrightarrow{\sqsubseteq} \texttt{tt}}$$

Rule 40 (case-split).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \begin{pmatrix} \text{case } M \text{ of} \\ \quad P_1 \to A_1 \;... \\ \quad P_n \to A_n \end{pmatrix} \sqsubseteq B \xrightarrow{\mathfrak{R}} \begin{pmatrix} \text{case } M \text{ of} \\ \quad P_1 \to A_1 \sqsubseteq B \;... \\ \quad P_n \to A_n \sqsubseteq B \end{pmatrix}}$$

This rule turns a pattern-match on the left-hand side of $\sqsubseteq$ into a case-analysis. That this rewrite preserves denotational equality relies on the fact that case-splitting returns `tt` if the case-split term M is undefined. It is worth noting that, unlike other automated theorem provers, Elea does not require a heuristic to decide which terms within a property to case-split upon. Elea will case-split on any pattern match which occurs topmost on the left-hand side of an approximation.

Rule 41 (left transitivity).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_{+} A'}{\Gamma, \Phi, \mathcal{H} \vdash A \sqsubseteq B \xrightarrow{\mathfrak{R}} A' \sqsubseteq B}$$

This rule recursively applies $\xrightarrow{\mathfrak{R}}_{+}$ to the left-hand side of $\sqsubseteq$. If $\mathfrak{R} = \sqsubseteq$ we have $A \sqsubseteq A'$ and so $A' \sqsubseteq B \Rightarrow A \sqsubseteq B$, meaning $(A \sqsubseteq B) \sqsubseteq (A' \sqsubseteq B)$. If $\mathfrak{R} = \sqsupseteq$ we have $A' \sqsubseteq A$ and so $A \sqsubseteq B \Rightarrow A' \sqsubseteq B$, meaning $(A \sqsubseteq B) \sqsupseteq (A' \sqsubseteq B)$

Rule 42 (right transitivity).

$$\frac{\Gamma, \Phi, \mathcal{H} \vdash B \xrightarrow{\mathfrak{R}^{-1}}_{+} B'}{\Gamma, \Phi, \mathcal{H} \vdash A \sqsubseteq B \xrightarrow{\mathfrak{R}} A \sqsubseteq B'}$$

Defining $\sqsubseteq^{-1} = \sqsupseteq$ and $\sqsupseteq^{-1} = \sqsubseteq$, we can use this rule to rewrite the right-hand side of $\sqsubseteq$. In proving $A \sqsubseteq B$ we will have $\mathfrak{R} = \sqsubseteq$, so this rule will mostly be used to apply $\xrightarrow{\sqsupseteq}_{+}$ to the right-hand side of a $\sqsubseteq$ property. It is for this rule that the $\xrightarrow{\sqsupseteq}_{+}$ rewrite exists.

Rule 43 (left forall).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash (\text{fn } x : \tau .\, A) \sqsubseteq B \xrightarrow{\mathfrak{R}} \text{forall } x : \tau .\, (A \sqsubseteq B\, x)}$$
$$\text{if} \quad x \notin \mathsf{freeVars}\,(B)$$

With this rule and right forall below we can prove properties between function terms.

Rule 44 (right forall).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash A \sqsubseteq (\text{fn } x : \tau .\, B) \xrightarrow{\mathfrak{R}} \text{forall } x : \tau .\, (A\, x \sqsubseteq B)}$$
$$\text{if} \quad x \notin \mathsf{freeVars}\,(A)$$

The left forall rule applied to the right-hand side of $\sqsubseteq$.

Rule 45 (traverse forall).

$$\frac{\Gamma[x \mapsto \tau], \Phi, \mathcal{H} \vdash \mathcal{P} \xrightarrow{\mathfrak{R}} \mathcal{Q}}{\Gamma, \Phi, \mathcal{H} \vdash \text{forall } x : \tau .\, \mathcal{P} \xrightarrow{\mathfrak{R}} \text{forall } x : \tau .\, \mathcal{Q}}$$

Rule 46 (least fixed-point).

$$\frac{}{\Gamma, \Phi, \mathcal{H} \vdash \text{fix}\,(F)\, x_1 ... x_n \sqsubseteq B \xrightarrow{\sqsubseteq} F\, (\text{fn } x_1, ..., x_n .\, B)\, x_1 ... x_n \sqsubseteq B}$$
$$\begin{aligned} \text{if} \quad & \{\ x_1, ..., x_n\ \} \cap \mathsf{freeVars}\,(B) = \emptyset \\ & \forall (i \neq j)\,.\, x_i \neq x_j \end{aligned}$$

Here is the rule that motivated fixed-point promoted form as the goal of $\xrightarrow{\sqsubseteq}_{+}$, since fixed-point promoted form is exactly those terms which, if appearing on the left-hand side of $\sqsubseteq$, allow us to perform this rewrite. This theorem proving process is to use left transitivity to apply $\xrightarrow{\sqsubseteq}_{+}$ to the left-hand side of $\sqsubseteq$ until we can apply this rule. We can then repeat this process until it can apply a rule like reflexivity or bottom. This is explained at a higher level in Chapter 3.

7.3 Example proofs

Having described the $\nu\text{PCF}^{\sqsubseteq}$ language, the rewrites applicable to it, and that $\overset{\sqsubseteq}{\longrightarrow}_+$ is a top-down theorem prover over it, this section gives some examples of this proof-by-rewriting process.

Example 7.1. A proof that the `eq` function, an equality predicate for natural numbers (defined in Appendix A), is symmetric.

$$\texttt{eq}\,x\,y \sqsubseteq \texttt{eq}\,y\,x$$

$$\overset{\sqsubseteq}{\longrightarrow} \{\text{ by least pre-fixed-point }\}$$

$$\begin{pmatrix}\text{fn } f, x, y.\ \text{case } x, y \text{ of} & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \texttt{True} \\ \quad \texttt{0}, \texttt{Suc}\,y' & \rightarrow & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{0} & \rightarrow & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{Suc}\,y' & \rightarrow & f\,x'\,y'\end{pmatrix} (\text{fn } x, y.\ \texttt{eq}\,y\,x)\,x\,y \ \sqsubseteq\ \texttt{eq}\,y\,x$$

$$\overset{\sqsubseteq}{\longrightarrow}_+ \{\text{ by beta reduction }\}$$

$$\begin{pmatrix}\text{case } x, y \text{ of} & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \texttt{True} \\ \quad \texttt{0}, \texttt{Suc}\,y' & \rightarrow & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{0} & \rightarrow & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{Suc}\,y' & \rightarrow & \texttt{eq}\,y'\,x'\end{pmatrix} \ \sqsubseteq\ \texttt{eq}\,y\,x$$

$$\overset{\sqsubseteq}{\longrightarrow} \{\text{ by case-split }\}$$

$$\begin{array}{lllll}\text{case } x, y \text{ of} & & & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \texttt{True} & \sqsubseteq & \texttt{eq}\,y\,x \\ \quad \texttt{0}, \texttt{Suc}\,y' & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{eq}\,y\,x \\ \quad \texttt{Suc}\,x', \texttt{0} & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{eq}\,y\,x \\ \quad \texttt{Suc}\,x', \texttt{Suc}\,y' & \rightarrow & \texttt{eq}\,y'\,x' & \sqsubseteq & \texttt{eq}\,y\,x\end{array}$$

$$\overset{\sqsubseteq}{\longrightarrow}_+ \{\text{ by case-var substitution }\}$$

$$\begin{array}{lllll}\text{case } x, y \text{ of} & & & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \texttt{True} & \sqsubseteq & \texttt{eq}\,\texttt{0}\,\texttt{0} \\ \quad \texttt{0}, \texttt{Suc}\,y' & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{eq}\,(\texttt{Suc}\,y')\,\texttt{0} \\ \quad \texttt{Suc}\,x', \texttt{0} & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{eq}\,\texttt{0}\,(\texttt{Suc}\,x') \\ \quad \texttt{Suc}\,x', \texttt{Suc}\,y' & \rightarrow & \texttt{eq}\,y'\,x' & \sqsubseteq & \texttt{eq}\,(\texttt{Suc}\,y')\,(\texttt{Suc}\,x')\end{array}$$

$$\overset{\sqsubseteq}{\longrightarrow}_+ \{\text{ by unfold-fix }\}$$

$$\begin{array}{lllll}\text{case } x, y \text{ of} & & & & \\ \quad \texttt{0}, \texttt{0} & \rightarrow & \texttt{True} & \sqsubseteq & \texttt{True} \\ \quad \texttt{0}, \texttt{Suc}\,y' & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{0} & \rightarrow & \texttt{False} & \sqsubseteq & \texttt{False} \\ \quad \texttt{Suc}\,x', \texttt{Suc}\,y' & \rightarrow & \texttt{eq}\,y'\,x' & \sqsubseteq & \texttt{eq}\,y'\,x'\end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by reflexivity }

$$\text{case } x, y \text{ of} \quad \begin{array}{lcl} \mathtt{0}, \mathtt{0} & \to & \mathtt{tt} \\ \mathtt{0}, \mathtt{Suc}\, y' & \to & \mathtt{tt} \\ \mathtt{Suc}\, x', \mathtt{0} & \to & \mathtt{tt} \\ \mathtt{Suc}\, x', \mathtt{Suc}\, y' & \to & \mathtt{tt} \end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by constant-case }

`tt`

Example 7.2. A proof that the tail-recursive definition of addition, `it-add`, approximates the non-tail-recursive definition, `add`. All terms are defined in Appendix A.

$\mathtt{it\text{-}add}\, x\, y \sqsubseteq \mathtt{add}\, x\, y$

$\xrightarrow{\sqsubseteq}$ { by constant argument fusion }

$\mathtt{it\text{-}add}\, x\, y \sqsubseteq \mathtt{add}_y\, x$

$\xrightarrow{\sqsubseteq}$ { by least pre-fixed-point }

$$\left(\begin{array}{l} \text{fn } f, x, y.\ \text{case } x \text{ of} \\ \quad \begin{array}{lcl} \mathtt{0} & \to & y \\ \mathtt{Suc}\, x' & \to & f\, x'\, (\mathtt{Suc}\, y) \end{array} \end{array}\right) (\text{fn } x, y.\ \mathtt{add}_y\, x)\, x\, y \ \sqsubseteq\ \mathtt{add}_y\, x$$

$\xrightarrow{\sqsubseteq}_{+}$ { by beta reduction }

$$\left(\begin{array}{l} \text{case } x \text{ of} \\ \quad \begin{array}{lcl} \mathtt{0} & \to & y \\ \mathtt{Suc}\, x' & \to & \mathtt{add}_{(\mathtt{Suc}\, y)}\, x' \end{array} \end{array}\right) \ \sqsubseteq\ \mathtt{add}_y\, x$$

$\xrightarrow{\sqsubseteq}_{+}$ { by case-split and case-var substitution }

$$\text{case } x \text{ of} \quad \begin{array}{lcl} \mathtt{0} & \to & y \sqsubseteq \mathtt{add}_y\, \mathtt{0} \\ \mathtt{Suc}\, x' & \to & \mathtt{add}_{(\mathtt{Suc}\, y)}\, x' \sqsubseteq \mathtt{add}_y\, (\mathtt{Suc}\, x') \end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by unfold fix }

$$\text{case } x \text{ of} \quad \begin{array}{lcl} \mathtt{0} & \to & y \sqsubseteq y \\ \mathtt{Suc}\, x' & \to & \mathtt{add}_{(\mathtt{Suc}\, y)}\, x' \sqsubseteq \mathtt{Suc}\, (\mathtt{add}_y\, x') \end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by constructor fission }

$$\text{case } x \text{ of} \quad \begin{array}{lcl} \mathtt{0} & \to & y \sqsubseteq y \\ \mathtt{Suc}\, x' & \to & \mathtt{Suc}\, (\mathtt{add}_y\, x') \sqsubseteq \mathtt{Suc}\, (\mathtt{add}_y\, x') \end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by reflexivity and constant-case }

`tt`

Chapter 8

Discovering fold functions

The previous chapter described how we use $\xrightarrow{\sqsubseteq}_{+}$ as a top down theorem prover over our property language $\nu\text{PCF}^{\sqsubseteq}$. In this chapter I give a technique in which this theorem prover is used within itself to perform a new rewriting technique, something I call fold discovery. This technique has greatly extended the rewriting capabilities, and hence the proving power, of Elea. It enables, for example, the rewrites

$$\begin{aligned}
&\texttt{sorted}\,(\texttt{isort}\,xs) && \xrightarrow{\sqsubseteq}_{+} && \texttt{True} \\
&\texttt{it-rev}\,xs\,\texttt{[ ]} && \xrightarrow{\sqsubseteq}_{+} && \texttt{rev}\,xs \\
&\texttt{map}_f\,(\texttt{iterate}_f\,x) && \xrightarrow{\sqsubseteq}_{+} && \texttt{iterate}_{f^2}\,x
\end{aligned}$$

This chapter is broken down as follows.

(Section 8.1)	Motivating examples which demonstrate how fold discovery works at a high-level.
(Section 8.3)	The fold discovery rewrite rule.
(Section 8.4)	The examples given above Section 8.1, formally described.

8.1 Motivating examples

Before describing this method in detail I will show where it is necessary, and how it works at a high level. Let's say we are tasked with proving

$$\texttt{it-rev}\,xs\,\texttt{[ ]} \sqsubseteq \texttt{rev}\,xs \qquad \text{(terms defined in Appendix A)}$$

To invoke the least fixed-point rule we have to rewrite `it-rev` xs `[ ]` to fixed-point promoted form. Since we are aiming for FPPF we need a fusion rule (Chapter 5), and the one that matches the shape of our term is accumulation fusion. In performing the antecedent rewrite within accumulation fusion we get stuck on the following term:

$$\begin{array}{l}
\text{case } xs \text{ of} \\
\quad \begin{array}{lcl}
\texttt{[ ]} & \to & \texttt{[ ]} \\
x :: xs' & \to & \texttt{it-rev}^{a}\,xs'\,[x]
\end{array}
\end{array}$$

The folding rewrite we are trying to apply to the above is:

$$(\text{fn } xs.\, \texttt{it-rev}^a\, xs\, \texttt{[ ]}) \sqsubseteq h$$

To apply this our tool needs to find some term F such that:

$$\texttt{it-rev}^a\, xs'\, [x] \xrightarrow{\sqsubseteq} F\, (\texttt{it-rev}^a\, xs'\, \texttt{[ ]})$$

As you might have guessed from the name, fold discovery discovers fold functions, which is to say functions definable using the $\text{fold}_{\boldsymbol{T}}\langle ... \rangle$ syntax (page 25). It works by assuming the term F is such a function, so for some value of c_1 and c_2:

$$\texttt{it-rev}^a\, xs'\, [x] \sqsubseteq \text{fold}_{\texttt{List}}\langle c_1, c_2\rangle\, (\texttt{it-rev}^a\, xs'\, \texttt{[ ]})$$

Now, instead of solving for F we are solving for c_1 and c_2. To do this, fold discovery uses $\xrightarrow{\sqsubseteq}_+$ to rewrite the property above, simplifying it to the point that the values of c_1 and c_2 become obvious. This process is given fully in Example 8.1 but for now I will just show the resulting property below.

$$\text{fix}^a \begin{pmatrix} \text{fn } h, xs, ys. \\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to \\ \qquad h\, xs'\, (x :: ys) \end{pmatrix} xs'\, [x] \sqsubseteq \text{fix}^a \begin{pmatrix} \text{fn } h, xs, ys. \\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to \\ \qquad h\, xs'\, (c_2\, x\, ys) \end{pmatrix} xs\, c_1$$

We can easily see that the above will be satisfied by $c_1 = [x]$ and $c_2 = \texttt{Cons}$. Since $\xrightarrow{\sqsubseteq}_+$ rewrites properties into sufficient properties, these values will also solve our original approximation. Since $\text{fold}_{\texttt{List}}\langle [x], \texttt{Cons}\rangle =_\alpha \texttt{snoc}_x$ this process has discovered

$$\texttt{it-rev}^a\, xs'\, [x] \sqsubseteq \texttt{snoc}_x\, (\texttt{it-rev}^a\, xs'\, \texttt{[ ]})$$

So it can rewrite the term we are stuck on to:

$$\begin{array}{l} \text{case } xs \text{ of} \\ \quad \begin{array}{lcl} \texttt{[ ]} & \to & \texttt{[ ]} \\ x :: xs' & \to & \texttt{snoc}_x\, (\texttt{it-rev}^a\, xs'\, \texttt{[ ]}) \end{array} \end{array}$$

Allowing us to apply folding rewrite required:

$$\begin{array}{l} \text{case } xs \text{ of} \\ \quad \begin{array}{lcl} \texttt{[ ]} & \to & \texttt{[ ]} \\ x :: xs' & \to & \texttt{snoc}_x\, (h\, xs') \end{array} \end{array}$$

This term is α-equal to the body of the `rev` function, so our accumulation fusion step has rewritten $\texttt{it-rev}\, xs\, \texttt{[ ]} \xrightarrow{\sqsubseteq} \texttt{rev}\, xs$. This process of inserting term meta-variables based on the shape we are attempting to rewrite a term, and later discovering their definitions, is referred to as middle-out reasoning, and was originally used in the Oyster/Clam theorem prover [30]. See Section 10.2.4 on page 138 for a detailed comparison.

Another example requiring fold discovery is our proof of:

$$\texttt{sorted}\,(\texttt{isort}\,xs) \sqsubseteq \texttt{True}$$

Again we are searching for a fixed-point promoted form for the left-hand side. This time our applicable rule is fix-fix fusion and within this step we reach the term:

```
case $xs$ of
  $[\,]$        $\to$  $\texttt{True}$
  $x :: xs'$   $\to$  $\texttt{sorted}\,(\texttt{insert}_x\,(\texttt{isort}^a\,xs'))$
```

The $\texttt{insert}_x$ function (defined in Appendix A) inserts the number x into a list while preserving sortedness. Our tool will run fix-fix fusion again on the above term, yielding

```
case $xs$ of
  $[\,]$        $\to$  $\texttt{True}$
  $x :: xs'$   $\to$  $\texttt{sorted-insert}_x\,(\texttt{isort}^a\,xs')$
```

Which is where we get stuck again. The fusion fact we are aiming to apply is

$$(\text{fn } xs.\, \texttt{sorted}\,(\texttt{isort}^a\,xs)) \sqsubseteq h$$

So we need to discover some term F such that

$$\texttt{sorted-insert}_x\,(\texttt{isort}^a\,xs') \sqsubseteq F\,(\texttt{sorted}\,(\texttt{isort}^a\,xs'))$$

To simplify our examples we generalise $\texttt{isort}^a\,xs'$ to ys now, a generalisation that the fold-discovery step will automatically perform:

$$\texttt{sorted-insert}_x\,ys \sqsubseteq F\,(\texttt{sorted}\,ys)$$

As in the previous example we assume F is a fold:

$$\texttt{sorted-insert}_x\,ys \sqsubseteq \text{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\,(\texttt{sorted}\,ys)$$

The fold discovery rule then uses $\xrightarrow{\sqsubseteq}_+$ to rewrite the above approximation. This rewrite is given in Example 8.3 on page 117 but the result is below.

```
case $ys$ of
  $[\,] \to \texttt{True} \sqsubseteq c_1$
  $x :: xs' \to$
    if $\texttt{lq}\,n\,x$
    then if $\texttt{sorted}\,(x :: xs')$
      then $\texttt{True} \sqsubseteq c_1$
      else $\texttt{False} \sqsubseteq c_2$
    else
      case $xs'$ of
        $[\,] \to \texttt{True} \sqsubseteq c_1$
        $x' :: xs'' \to$
          if $\texttt{lq}\,n\,x'$
          then $\texttt{tt}$
          else if $\texttt{lq}\,x\,x'$
            then $\texttt{tt}$
            else $\texttt{False} \sqsubseteq c_2$
```

As you can see, the above property will be solved if $c_1 = \texttt{True}$ and $c_2 = \texttt{False}$. Since $\xrightarrow{\sqsubseteq}_+$ preserves $\Leftarrow$, meaning that if $\mathcal{P} \xrightarrow{\sqsubseteq}_+ \mathcal{Q}$ then we have that $\mathcal{Q} \Rightarrow \mathcal{P}$, these values will also solve our original approximation. This is to say that if we set $c_1 = \texttt{True}$ and $c_2 = \texttt{False}$, then the above property will hold, and hence so will the original property:

$$\texttt{sorted-insert}_x\, ys \sqsubseteq \text{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\ (\texttt{sorted}\, ys)$$

We have therefore discovered that:

$$\texttt{sorted-insert}_x\, ys \sqsubseteq \text{fold}_{\texttt{Bool}}\langle \texttt{True}, \texttt{False}\rangle\ (\texttt{sorted}\, ys)$$

The rule identity fission can then rewrite the right-hand side since $\text{fold}_{\texttt{Bool}}\langle \texttt{True}, \texttt{False}\rangle$ is just an identity function on booleans.

$$\texttt{sorted-insert}_x\, ys \sqsubseteq \texttt{sorted}\, ys$$

This allows us to rewrite the term we are stuck on to

$$\begin{array}{l} \text{case } xs \text{ of} \\ \quad \begin{array}{lcl} \texttt{[]} & \to & \texttt{True} \\ x :: xs' & \to & \texttt{sorted}\,(\texttt{isort}^a\, xs') \end{array} \end{array}$$

So we can perform the folding step, yielding

$$\begin{array}{l} \text{case } xs \text{ of} \\ \quad \begin{array}{lcl} \texttt{[]} & \to & \texttt{True} \\ x :: xs' & \to & h\, xs' \end{array} \end{array}$$

The fix-fix fusion rule originally applied has now rewritten

$$\texttt{sorted}\,(\texttt{isort}\, xs) \xrightarrow{\sqsubseteq} \text{fix} \left(\begin{array}{l} \text{fn } h, xs.\ \text{case } xs \text{ of} \\ \quad \begin{array}{lcl} \texttt{[]} & \to & \texttt{True} \\ x :: xs' & \to & h\, xs' \end{array} \end{array} \right) xs$$

Sub-term fission can rewrite right-hand side of the above to $\texttt{True}$, so to summarise fold discovery has allowed us to rewrite

$$\texttt{sorted}\,(\texttt{isort}\, xs) \xrightarrow{\sqsubseteq}_+ \texttt{True}$$

This section has given two examples of how fold discovery works at a high level. In both examples we were able to discover a value for both c_1 and c_2 in our fold function. The next section gives an example in which we can only find a value for c_1 but not c_2, and explains how we can deal with a partial solution like this.

8.2 Partial solutions

The previous section gave two examples of how fold discovery works at a high-level. The second of these found values for c_1 and c_2 such that

$$\texttt{sorted-insert}_x\, ys \sqsubseteq \mathrm{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\ (\texttt{sorted}\, ys)$$

This section describes what Elea does when it can only find a partial solution to its fold discovery approximation, for example if we could only find a value for c_1 but not for c_2. Such an example is to be found in rewriting the boolean term

$$\texttt{le}\,(\texttt{len}\,(\texttt{filter}_p\, xs))\ (\texttt{len}\, xs)$$

The above calculates whether the length of a filtered list is less-than-or-equal to the length of the list. After a couple of fusion rewrites Elea produces the equivalent term below, where `le-len` checks whether the length of its first list argument is less-than-or-equal to the length of the second.

$$\texttt{le-len}\,(\texttt{filter}_p\, xs)\ xs$$

In attempting fix-fix fusion on the above, Elea reaches the following term

$$\begin{array}{l}
\text{case } xs \text{ of}\\
\quad \begin{array}{lll}
\texttt{[]} & \rightarrow & \texttt{True}\\
x :: xs' & \rightarrow & \text{if } p\, x\\
& & \begin{array}{ll}\text{then} & \texttt{le-len}\,(\texttt{filter}^a_p\, xs')\ xs'\\
\text{else} & \texttt{le-len}\,(\texttt{filter}^a_p\, xs')\ (x :: xs')\end{array}
\end{array}
\end{array}$$

The folding rewrite Elea will be attempting to apply is

$$(\text{fn } xs.\, \texttt{le-len}\,(\texttt{filter}^a_p\, xs)\ xs) \sqsubseteq h$$

Which it can do when $p\,x$ holds, but not otherwise, leaving us with the following term.

$$\begin{array}{l}
\text{case } xs \text{ of}\\
\quad \begin{array}{lll}
\texttt{[]} & \rightarrow & \texttt{True}\\
x :: xs' & \rightarrow & \text{if } p\, x\\
& & \text{then } h\, xs'\\
& & \text{else } \texttt{le-len}\,(\texttt{filter}^a_p\, xs')\ (x :: xs')
\end{array}
\end{array}$$

Enter fold discovery, which will generate this approximation to be solved for c_1 and c_2.

$$\texttt{le-len}\,(\texttt{filter}^a_p\, xs')\ (x :: xs') \sqsubseteq \mathrm{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\ (\texttt{le-len}\,(\texttt{filter}^a_p\, xs')\ xs')$$

The rewrite rule generalise argument will simplify this to the following, for some fresh ys.

$$\texttt{le-len}\, ys\ (x :: xs') \sqsubseteq \mathrm{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\ (\texttt{le-len}\, ys\ xs')$$

Elea can rewrite fold functions over booleans into just if expressions, so the above becomes

$$\texttt{le-len}\, ys\ (x :: xs') \sqsubseteq \text{if } \texttt{le-len}\, ys\, xs' \text{ then } c_1 \text{ else } c_2$$

Rewriting this property fully using $\xrightarrow{\sqsubseteq}_{+}$, as given in Example 8.4 on page 119, yields

```
case $ys, xs'$ of
  [ ], [ ]              →  True ⊑ $c_1$
  [ ], $x :: xs'$        →  True ⊑ $c_1$
  $y :: ys'$, [ ]        →  null $ys'$ ⊑ $c_2$
  $y :: ys', x' :: xs''$  →  tt
```

From this we can discover the value of c_1 as `True`, since if the length of a list ys is less-than-or-equal to xs', then adding an element to the head of the larger list will preserve this property. Unfortunately there is no value of c_2 which solves the above! This is because adding an element to the head of a smaller list might break the property, so c_2 will be `False` if the length of xs' is one less than that of ys, but `True` otherwise.

It turns out that a partial solution like this can be helpful, we just need a "default" value for c_2 to solve the approximation. If the data-type we are folding over is non-recursive then there exists such a value, the one we are attempting to rewrite, which in this example is `le-len` $ys\ (x :: xs')$. So, choosing $c_2 =$ `le-len` $ys\ (x :: xs')$ we have the following.

$$\texttt{le-len}\, ys\ (x :: xs') \sqsubseteq \text{if } \texttt{le-len}\, ys\, xs' \text{ then } \texttt{True} \text{ else } \texttt{le-len}\, ys\ (x :: xs')$$

Ungeneralising ys back to $\texttt{filter}^a_p\, xs'$ in the above allows Elea to rewrite the term it was stuck on to

```
case $xs$ of
  [ ]       →  True
  $x :: xs'$  →  if $p\,x$
               then $h\,xs'$
               else if le-len ($\texttt{filter}^a_p\,xs'$) $xs'$
                 then True
                 else le-len ($\texttt{filter}^a_p\,xs'$) $(x :: xs')$
```

The folding rewrite fn xs. `le-len` $(\texttt{filter}^a_p\, xs)\, xs \sqsubseteq h$ is now applicable to the `not` $(p\,x)$ branch, yielding

```
case $xs$ of
  [ ]       →  True
  $x :: xs'$  →  if $p\,x$
               then $h\,xs'$
               else if $h\,xs'$
                 then True
                 else le-len ($\texttt{filter}^a_p\,xs'$) $(x :: xs')$
```

So, with the folding rewrite applied, our fix-fix fusion antecedent rewrite finishes. The full rewrite Elea performs is as follows:

$$\texttt{le}\,(\texttt{len}\,(\texttt{filter}_p\,xs))\,(\texttt{len}\,xs)$$

$$\xrightarrow{\sqsubseteq}_+ \{ \text{ by two fix-fix fusion rewrites } \}$$

$$\texttt{le-len}\,(\texttt{filter}_p\,xs)\,xs$$

$$\xrightarrow{\sqsubseteq} \{ \text{ by fix-fix fusion using the fold discovery step as just explained } \}$$

$$\text{fix}\left(\begin{array}{l}\text{fn } h, xs.\,\text{case } xs \text{ of}\\ \quad [\,] \to \texttt{True}\\ \quad x :: xs' \to\\ \quad\quad \text{if } \ p\,x\\ \quad\quad \text{then } h\,xs'\\ \quad\quad \text{else if } h\,xs'\\ \quad\quad\quad \text{then } \texttt{True}\\ \quad\quad\quad \text{else } \texttt{le-len}\,(\texttt{filter}^a_p\,xs')\,(x :: xs')\end{array}\right) xs$$

$$\xrightarrow{\sqsubseteq}_+ \{ \text{ by subterm fission of fn } xs.\,\texttt{True} \text{ then beta reduction } \}$$

$$\texttt{True}$$

The final subterm fission step above would not have been applicable had we not performed fold discovery with a partial solution.

8.3 Fold discovery

Prior in this chapter I informally explained how fold discovery can find values for c_1 and c_2 which solve the following approximations.

$$\begin{array}{lcl}\texttt{it-rev}^a\,xs'\,[x] & \sqsubseteq & \text{fold}_{\texttt{List}}\langle c_1, c_2\rangle\,(\texttt{it-rev}^a\,xs'\,[\,])\\ \texttt{sorted-insert}_x\,ys & \sqsubseteq & \text{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\,(\texttt{sorted}\,ys)\\ \texttt{le-len}\,ys\,(x :: xs) & \sqsubseteq & \text{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\,(\texttt{le-len}\,ys\,xs)\end{array}$$

Here, $\texttt{sorted-insert}_x$ is the result of fusing the expression fn $xs.\,\texttt{sorted}\,(\texttt{insert}_x\,xs)$, and `le-len` is the result of fusing fn $xs, ys.\,\texttt{le}\,(\texttt{len}\,xs)\,(\texttt{len}\,ys)$.

Discovering c_1 and c_2 in the above done by using our top down theorem prover $\xrightarrow{\sqsubseteq}_+$ (Chapter 7) to rewrite these approximations, after which we should hopefully be left with a property from which we can easily deduce values of c_1 and c_2. The previous section explained how in the final example above we are only able to deduce the value of c_1, but that this is okay since we can use $\texttt{le}\,(\texttt{len}\,ys)\,(\texttt{len}\,(x :: xs))$ as the value for c_2.

This section gives the formal rewrite rule which performs this process, fold discovery. It relies on two auxiliary functions, guessArgs and solve. The aim of fold discovery is to allow the application of a folding rewrite stored in our fact environment Φ. Folding rewrites take a list of arguments, and we need to guess the value of these arguments in order to perform fold discovery. It is the guessArgs operator, given in Definition 8.1 which guesses these arguments. There are no properties guessArgs needs to fulfil, as the fold discovery rule has a correctness check built into it.

The solve operator takes a variable c, and a νPCF$^{\sqsubseteq}$ property term $\mathcal{P}$, and returns the term we should substitute for c in $\mathcal{P}$ if we wish to make the property hold. Definition 8.1 gives this operator, and as with guessArgs it does not need obey any specification for fold discovery to be sound, as this rewrite explicitly checks its own soundness.

Definition 8.1 (guessArgs).

$$\mathsf{guessArgs}(\text{fn } x_1, ..., x_n.\, \mathcal{C}, \mathcal{C}') = A_1, ..., A_n$$

$$\begin{aligned} \text{if} \quad & f \text{ is a fresh variable} \\ & \forall((f\ B_1 ... B_n) \in \mathsf{subterms}(\mathcal{C}[f]))\ . \\ & \quad \forall((f\ B'_1 ... B'_n) \in \mathsf{subterms}(\mathcal{C}'[f]))\ . \\ & \quad \forall(i \leq n, j \leq n)\ .\ B_i = x_j \Rightarrow A_j = B'_i \end{aligned}$$

The guessArgs operator takes two contexts in which the context hole is a function, and returns the terms which if applied as arguments to the first context will make the arguments to the context hole in both contexts match. It's return value is not defined if no such arguments exist. In my implementation of this function the values for $A_1 ... A_n$ are selected by enumerating all $(f\ B_1 ... B_n) \in \mathsf{subterms}(\mathcal{C}[f])$ and $(f\ B'_1 ... B'_n) \in \mathsf{subterms}(\mathcal{C}'[f])$, and choosing values such that $B_i = x_j \Rightarrow A_j = B'_i$ holds, viz. if $B_i = x_j$ for any i and j, then choose $A_j = B'_i$.

Below are some examples of using guessArgs:

$$\mathsf{guessArgs}(\text{fn } xs.\, \square\, xs\ \texttt{[ ]},\ \square\, xs'\ \texttt{[}x\texttt{]}) \ = \ xs'$$

$$\mathsf{guessArgs}(\text{fn } xs.\, \texttt{sorted}\, (\square\, xs),\ \texttt{sorted-insert}_x\, (\square\, xs')) \ = \ xs'$$

$$\mathsf{guessArgs}(\text{fn } xs.\, \texttt{le-len}\, (\square\, xs)\, xs,\ \texttt{le-len}\, (\square\, xs')\, (x \texttt{ :: } xs')) \ = \ xs'$$

Definition 8.2 (solve).

$$\mathsf{solve}\ c\ \mathcal{P} \ \overset{\text{def}}{=} \ \bot$$
$$\text{if} \quad c \notin \mathsf{freeVars}\,(\mathcal{P})$$

$$\mathsf{solve}\ c\ (A \sqsubseteq B) \ \overset{\text{def}}{=} \ c\,\sigma$$
$$\text{if} \quad A\sigma =_\alpha B\sigma \ \wedge \ c \in \mathsf{dom}\,(\sigma)$$

$$\mathsf{solve}\ c\ (\text{forall } x.\, \mathcal{P}) \ \overset{\text{def}}{=} \ \mathsf{solve}\ c\ \mathcal{P}$$

$$\mathsf{solve}\ c\ (\text{case } M \text{ of } P_1 \to \mathcal{Q}_1 \ldots P_n \to \mathcal{Q}_n) \ \overset{\text{def}}{=} \ C$$
$$\text{if} \quad \forall(i \leq n)\,.\,\mathsf{solve}\ c\ P_i = C \ \vee \ \mathsf{solve}\ c\ P_i = \bot$$

Given a variable c and a νPCF$^{\sqsubseteq}$ term $\mathcal{P}$, the solve operator suggests a term which could be substituted for c in νPCF$^{\sqsubseteq}$ in order to make the property hold. This value does not have to be correct in order for fold discovery to be sound, since it has an explicit soundness check built in.

Rule 47 (fold discovery).

$$\frac{\Gamma, \Phi, \mathcal{H}' \vdash \mathcal{P} \xrightarrow{\sqsubseteq}_{+} \mathcal{Q} \qquad \Gamma, \Phi, \mathcal{H}' \vdash \mathcal{Q}[C_1/c_1]...[C_m/c_m] \xrightarrow{\sqsubseteq}_{+} \texttt{tt}}{\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C}[\text{fix}^a\,(G)] \xrightarrow{\sqsubseteq} \text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\;(h\;A_1...A_n)}$$

$$\begin{aligned}
\text{if}\quad & \mathcal{H}' = \mathcal{H},\, \mathcal{C}[\text{fix}^a\,(G)] \\
& (\text{fn}\; x_1, ..., x_n.\, \mathcal{C}'[\text{fix}^a\,(G)] \sqsubseteq h) \in \Phi \\
& A_1...A_n = \mathsf{guessArgs}(\text{fn}\; x_1, ..., x_n.\, \mathcal{C}, \mathcal{C}') \\
& \Gamma \vdash h\; A_1...A_n : \boldsymbol{T} \\
& \boldsymbol{T} \text{ has } m \text{ constructors} \\
& c_1, ..., c_m \text{ fresh} \\
& \mathcal{P} = \mathcal{C}[\text{fix}^a\,(G)] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle c_1, ..., c_m\rangle\; (\mathcal{C}'[\text{fix}^a\,(G)][A_1/x_1]...[A_n/x_n]) \\
& C_i = \begin{cases} \mathsf{solve}\; c_i\; \mathcal{Q} & \text{if } \mathsf{solve}\; c_i\; \mathcal{Q} \text{ is defined} \\ \mathcal{C}[\text{fix}^a\,(G)] & \text{if } \boldsymbol{T} \text{ is non-recursive} \end{cases} \qquad \{\text{ for all } i \in [0..m] \;\}
\end{aligned}$$

The goal of fold discovery is to apply a folding rewrite

$$\text{fn}\; x_1, ..., x_n.\, \mathcal{C}'[\text{fix}^a\,(G)] \sqsubseteq h$$

In order to do this, we must first find likely values for the arguments to the left-hand side of this rewrite using the guessArgs operator (Definition 8.1). We then assume that the term we are rewriting approximates the the left-hand side of that rewrite, given the argument values for $x_1...x_n$ we found using guessArgs, and with a fold function surrounding it, for some value of variables $c_1, ..., c_m$, which we express as an $\nu\text{PCF}^{\sqsubseteq}$ term $\mathcal{P}$.

$$\mathcal{P} = \mathcal{C}[\text{fix}^a\,(G)] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle c_1, ..., c_m\rangle\; (\mathcal{C}'[\text{fix}^a\,(G)][A_1/x_1]...[A_n/x_n])$$

Running $\xrightarrow{\sqsubseteq}_{+}$ on this property yields sufficient property $\mathcal{Q}$, using solve on this property for every $c_1...c_m$ variable can hopefully find terms $C_1...C_m$ which we can substitute for $c_1...c_m$ in $\mathcal{Q}$, such that $\mathcal{Q}$ holds.

Since we have no guarantee from solve that the terms $C_1...C_m$ will actually solve $\mathcal{Q}$, we have to explicitly check that $\mathcal{Q}$ with these terms in place of $c_1...c_m$ makes the property true. As $\xrightarrow{\sqsubseteq}_{+}$ preserves backwards implication, we know that this substitution will also make $\mathcal{P}$ hold, hence we have

$$\mathcal{C}[\text{fix}^a\,(G)\, A_1...A_n] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\; (\mathcal{C}'[\text{fix}^a\,(G)]\; A_1...A_n)$$

So far, we have that the second antecedent rewrite asserts that the above holds. Now, we can apply this fact we are using from Φ with $A_1...A_n$ as arguments, which is to say $\mathcal{C}'[\text{fix}^a\,(G)]A_1...A_n \sqsubseteq h\; A_1...A_n$ to rewrite the right-hand side of the above, yielding a property which proves the soundness of this method

$$\Gamma, \Phi, \mathcal{H} \vdash \mathcal{C}[\text{fix}^a\,(G)] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\; (h\; A_1...A_n)$$

This section has described the fold discovery rewrite rule, a rule for which Elea invokes itself twice, first as a proof simplifier in order to conjecture a potential rewrite, and then as a theorem prover, in order to check that its conjectured rewrite was correct. Fold discovery demonstrates the utility of building a $\sqsubseteq$ property prover using a term rewriting system which preserves $\sqsubseteq$, as we can invoke the theorem prover to check the soundness of rewrites which the theorem prover itself uses. The next section gives examples of using fold discovery rule.

8.4 Examples of fold discovery

With the fold discovery rewrite formally defined in the previous section, this section gives fully worked examples of this rewriting technique.

Example 8.1. Here is the first example from Section 8.1 in which we are trying to find values of c_1 and c_2 which makes the following approximation hold:

$$\texttt{it-rev}^a\, xs'\; \texttt{[}x\texttt{]} \sqsubseteq \text{fold}_{\texttt{List}}\langle c_1, c_2\rangle\; (\texttt{it-rev}^a\, xs'\; \texttt{[ ]})$$

Fold discovery does this by using our $\xrightarrow{\sqsubseteq}_+$ rewrite on the above approximation.

$$\texttt{it-rev}^a\, xs'\; \texttt{[}x\texttt{]} \sqsubseteq \text{fold}_{\texttt{List}}\langle c_1, c_2\rangle\; (\texttt{it-rev}^a\, xs'\; \texttt{[ ]})$$

$$\xrightarrow{\sqsubseteq}_+ \{ \text{ by Example 8.2 using right transitivity } \}$$

$$\texttt{it-rev}^a\, xs'\; \texttt{[}x\texttt{]} \sqsubseteq \text{fix}^a \begin{pmatrix} \text{fn } h, xs, ys. \\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to \\ \qquad h\, xs'\, (c_2\, x\, ys) \end{pmatrix} xs\; c_1$$

$$= \{ \text{ by definition of } \texttt{it-rev}^a \text{ (Appendix A) } \}$$

$$\text{fix}^a \begin{pmatrix} \text{fn } h, xs, ys. \\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to \\ \qquad h\, xs'\, (x :: ys) \end{pmatrix} xs'\; \texttt{[}x\texttt{]} \sqsubseteq \text{fix}^a \begin{pmatrix} \text{fn } h, xs, ys. \\ \text{case } xs \text{ of} \\ \quad \texttt{[ ]} \to ys \\ \quad x :: xs' \to \\ \qquad h\, xs'\, (c_2\, x\, ys) \end{pmatrix} xs\; c_1$$

Running solve on the above yields $c_1 = \texttt{[}x\texttt{]}$ and $c_2 = \texttt{Cons}$, meaning we have discovered

$$\texttt{it-rev}^a\, xs'\; \texttt{[}x\texttt{]} \sqsubseteq \text{fold}_{\texttt{List}}\langle \texttt{[}x\texttt{]}, \texttt{Cons}\rangle\; (\texttt{it-rev}^a\, xs'\; \texttt{[ ]})$$

Since $\text{fold}_{\texttt{List}}\langle \texttt{[}x\texttt{]}, \texttt{Cons}\rangle =_\alpha \texttt{snoc}_x$ we have

$$\texttt{it-rev}^a\, xs'\; \texttt{[}x\texttt{]} \sqsubseteq \texttt{snoc}_x\; (\texttt{it-rev}^a\, xs'\; \texttt{[ ]})$$

Example 8.2. This example is used within the above fold discovery example.

$$\mathrm{fold}_{\texttt{List}}\langle c_1, c_2\rangle\ (\texttt{it-rev}^a\ xs\ \texttt{[ ]})$$

$$\xrightarrow{\sqsupseteq} \{\text{ by fix-fix fusion }\}$$

$$\mathrm{fix}^a \begin{pmatrix} \text{fn } h, xs, ys.\ \text{case } xs \text{ of} & & \\ \quad \texttt{[ ]} & \to & \mathrm{fold}_{\texttt{List}}\langle c_1, c_2\rangle\ ys \\ \quad x :: xs' & \to & h\, xs'\,(x :: ys) \end{pmatrix}\ xs\ \texttt{[ ]}$$

$$\xrightarrow{\sqsupseteq} \{\text{ by accumulation fission }\}$$

$$\mathrm{fix}^a \begin{pmatrix} \text{fn } h, xs, ys.\ \text{case } xs \text{ of} & & \\ \quad \texttt{[ ]} & \to & ys \\ \quad x :: xs' & \to & h\, xs'\,(c_1\, x\, ys) \end{pmatrix}\ xs\ (\mathrm{fold}_{\texttt{List}}\langle c_1, c_2\rangle\ \texttt{[ ]})$$

$$\xrightarrow{\sqsupseteq} \{\text{ by unroll fix }\}$$

$$\mathrm{fix}^a \begin{pmatrix} \text{fn } h, xs, ys.\ \text{case } xs \text{ of} & & \\ \quad \texttt{[ ]} & \to & ys \\ \quad x :: xs' & \to & h\, xs'\,(c_1\, x\, ys) \end{pmatrix}\ xs\ c_2$$

Example 8.3. Here is the second example from Section 8.1 in which we are trying to discover values of c_1 and c_2 such that the following approximation holds:

$$\texttt{sorted-insert}_n\, ys \ \sqsubseteq\ \mathrm{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle(\texttt{sorted}\, ys)$$

The fold of a non-recursive data-type like `Bool` is a non-recursive function. Therefore, we can use the unfold fix rule to reduce it to just a pattern match.

$$\texttt{sorted-insert}_n\, ys \ \sqsubseteq\ \begin{pmatrix} \text{fn } ys.\ \text{case } \texttt{sorted}\, ys \text{ of} \\ \quad \texttt{True} \to c_1 \\ \quad \texttt{False} \to c_2 \end{pmatrix}\ ys$$

To stop terms getting too large I give the name `RHS` to the term on the right-hand side of the approximation.

$$\texttt{sorted-insert}_n\, ys \ \sqsubseteq\ \texttt{RHS}\, ys$$

Now, in our aim of discovering values for c_1 and c_2 which solve this approximation, we use our theorem proving $\xrightarrow{\sqsubseteq}_+$ rewrite on the above to simplify it.

$$\texttt{sorted-insert}_n\, ys \ \sqsubseteq\ \texttt{RHS}\, ys$$

$$\xrightarrow{\sqsubseteq} \{\text{ by least fixed-point rule }\}$$

$$\texttt{sorted-insert}'_n\, \texttt{RHS}\, ys \ \sqsubseteq\ \texttt{RHS}\, ys$$

$\xrightarrow{\sqsubseteq}_{+}$ { by definition of $\texttt{sorted-insert}'_n$, case-split rules and others }

```
case $ys$ of
  $[\,] \to \texttt{True} \sqsubseteq c_1$
  $x :: xs' \to$
    if $\texttt{lq}\, n\, x$
    then if $\texttt{sorted}\,(x :: xs')$
      then $\texttt{True} \sqsubseteq c_1$
      else $\texttt{False} \sqsubseteq c_2$
    else
      case $xs'$ of
        $[\,] \to \texttt{True} \sqsubseteq c_1$
        $x' :: xs'' \to$
          if $\texttt{lq}\, n\, x'$
          then if $\texttt{lq}\, x\, x'$
            then $\texttt{RHS}\,(x' :: xs'') \sqsubseteq \texttt{RHS}\,(x' :: xs'')$
            else $\texttt{RHS}\,(x' :: xs'') \sqsubseteq c_2$
          else if $\texttt{lq}\, x\, x'$
            then $\texttt{RHS}\,(x' :: xs'') \sqsubseteq \texttt{RHS}\,(x' :: xs'')$
            else $\texttt{False} \sqsubseteq c_2$
```

$\xrightarrow{\sqsubseteq}_{+}$ { by reflexivity }

```
case $ys$ of
  $[\,] \to \texttt{True} \sqsubseteq c_1$
  $x :: xs' \to$
    if $\texttt{lq}\, n\, x$
    thenif $\texttt{sorted}\,(x :: xs')$
      then $\texttt{True} \sqsubseteq c_1$
      else $\texttt{False} \sqsubseteq c_2$
    else
      case $xs'$ of
        $[\,] \to \texttt{True} \sqsubseteq c_1$
        $x' :: xs'' \to$
          if $\texttt{lq}\, n\, x'$
          thenif $\texttt{lq}\, x\, x'$
            then $\texttt{tt}$
            else $\texttt{RHS}\,(x' :: xs'') \sqsubseteq c_2$
          else if $\texttt{lq}\, x\, x'$
            then $\texttt{tt}$
            else $\texttt{False} \sqsubseteq c_2$
```

$\xrightarrow{\sqsubseteq}_+$ { by fact fusion then sub-term fission into case reduction }

```
case $ys$ of
  $[\,] \to \texttt{True} \sqsubseteq c_1$
  $x :: xs' \to$
    if $\texttt{lq}\, n\, x$
    then if $\texttt{sorted}\, (x :: xs')$
      then $\texttt{True} \sqsubseteq c_1$
      else $\texttt{False} \sqsubseteq c_2$
    else
      case $xs'$ of
        $[\,] \to \texttt{True} \sqsubseteq c_1$
        $x' :: xs'' \to$
          if $\texttt{lq}\, n\, x'$
          then $\texttt{tt}$
          else if $\texttt{lq}\, x\, x'$
            then $\texttt{tt}$
            else $\texttt{False} \sqsubseteq c_2$
```

The final $\xrightarrow{\sqsubseteq}_+$ rewrite in the above is fusing the fact that $\texttt{False} \sqsubseteq \texttt{lq}\, n\, x$ and $\texttt{True} \sqsubseteq \texttt{lq}\, n\, x'$ into the term $\texttt{lq}\, x\, x'$. To express this mathematically we are fusing the facts $n \not\leq x$ ($\Rightarrow x \leq n$) and $n \leq x'$ into $x \leq x'$. By transitivity we know that given these facts $x \leq x'$ will always be true, which is to say $\texttt{lq}\, x\, x'$ will always return `True`. This manifests in our match fusion step which rewrites $\texttt{lq}\, x\, x'$ into a fixed-point which only returns `True`. Subterm fission can then transform this fixed-point into the term `True` so case-reduction can remove the pattern match around it.

Now that we have a fully rewritten property we use the solve function on it to discover values $c_1 = \texttt{True}$ and $c_2 = \texttt{False}$ which make this property hold. Substituting these values in for c_1 and c_2 in the above property will allow it to be rewritten to `tt`, so we know these values are a correct solution. Since our property rewriting steps preserve $\Leftarrow$ we know that this substitution also satisfies our original inequality, so we have that

$$\texttt{sorted-insert}_n\, ys \;\sqsubseteq\; \text{fold}_{\texttt{Bool}}\langle \texttt{True}, \texttt{False}\rangle(\texttt{sorted}\, ys)$$

We can apply identity fission to the above to get the fully simplified approximation our fold discovery rule has found

$$\texttt{sorted-insert}_n\, ys \sqsubseteq \texttt{sorted}\, ys$$

Example 8.4. Here is the example from Section 8.2 in which we are trying to discover values of c_1 and c_2 such that the following approximation holds.

$$\texttt{le-len}\, ys\, (x :: xs) \sqsubseteq \text{fold}_{\texttt{Bool}}\langle c_1, c_2\rangle\, (\texttt{le-len}\, ys\, xs)$$

The fold of a non-recursive data-type like `Bool` is a non-recursive function. Therefore, we can use the unfold fix rule to reduce it to just a pattern match.

$$\texttt{le-len}\, ys\, (x :: xs) \sqsubseteq \begin{pmatrix} \text{fn } ys, xs.\ \text{case } \texttt{le-len}\, ys\, xs \text{ of} \\ \quad \texttt{True} \to c_1 \\ \quad \texttt{False} \to c_2 \end{pmatrix} ys\, xs$$

To stop terms getting too large I give the name RHS to the term on the right-hand side of the approximation.

$$\texttt{le-len}\, ys\ (x\ ::\ xs) \sqsubseteq\ \texttt{RHS}\, ys\ xs$$

Below is the result of fold discovery applying the theorem proving $\xrightarrow{\sqsubseteq}_{+}$ rewrite to this property.

$$\texttt{le-len}\, ys\ (x\ ::\ xs) \sqsubseteq\ \texttt{RHS}\, ys\ xs$$

$\xrightarrow{\sqsubseteq}$ { by constructor fusion }

$$\text{fix}\begin{pmatrix}\text{fn } h, ys, xs.\ \text{case } ys, xs \text{ of} & & \\ \quad \texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \\ \quad \texttt{[ ]}, x\ ::\ xs' & \rightarrow & \texttt{True} \\ \quad y\ ::\ ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \\ \quad y\ ::\ ys', x\ ::\ xs' & \rightarrow & h\, ys'\ xs'\end{pmatrix} ys\ xs \sqsubseteq\ \texttt{RHS}\, ys\ xs$$

$\xrightarrow{\sqsubseteq}$ { by least fixed-point principle }

$$\begin{pmatrix}\text{fn } h, ys, xs.\ \text{case } ys, xs \text{ of} & & \\ \quad \texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \\ \quad \texttt{[ ]}, x\ ::\ xs' & \rightarrow & \texttt{True} \\ \quad y\ ::\ ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \\ \quad y\ ::\ ys', x\ ::\ xs' & \rightarrow & h\, ys'\ xs'\end{pmatrix} (\text{fn } ys, xs.\ \texttt{RHS}\, ys\ xs) \sqsubseteq\ \texttt{RHS}\, ys\ xs$$

$\xrightarrow{\sqsubseteq}_{+}$ { by beta reduction and case-split }

$$\begin{array}{lll}\text{case } ys, xs \text{ of} & & \\ \quad \texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \sqsubseteq \texttt{RHS}\, ys\ xs \\ \quad \texttt{[ ]}, x\ ::\ xs' & \rightarrow & \texttt{True} \sqsubseteq \texttt{RHS}\, ys\ xs \\ \quad y\ ::\ ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \sqsubseteq \texttt{RHS}\, ys\ xs \\ \quad y\ ::\ ys', x\ ::\ xs' & \rightarrow & \texttt{RHS}\, ys'\ xs' \sqsubseteq \texttt{RHS}\, ys\ xs\end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by case-var substitution }

$$\begin{array}{lll}\text{case } ys, xs \text{ of} & & \\ \quad \texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \sqsubseteq \texttt{RHS [ ] [ ]} \\ \quad \texttt{[ ]}, x\ ::\ xs' & \rightarrow & \texttt{True} \sqsubseteq \texttt{RHS [ ]}\ (x\ ::\ xs') \\ \quad y\ ::\ ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \sqsubseteq \texttt{RHS}\ (y\ ::\ ys')\ \texttt{[ ]} \\ \quad y\ ::\ ys', x\ ::\ xs' & \rightarrow & \texttt{RHS}\, ys'\ xs' \sqsubseteq \texttt{RHS}\ (y\ ::\ ys')\ (x\ ::\ xs')\end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by definition of RHS and unfold-fix }

$$\begin{array}{lll}\text{case } ys, xs \text{ of} & & \\ \quad \texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \sqsubseteq c_1 \\ \quad \texttt{[ ]}, x\ ::\ xs' & \rightarrow & \texttt{True} \sqsubseteq c_1 \\ \quad y\ ::\ ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \sqsubseteq c_2 \\ \quad y\ ::\ ys', x\ ::\ xs' & \rightarrow & \texttt{RHS}\, ys'\ xs' \sqsubseteq \texttt{RHS}\, ys'\ xs'\end{array}$$

$\xrightarrow{\sqsubseteq}_{+}$ { by reflexivity }

$$\begin{array}{l}
\text{case } ys, xs \text{ of} \\
\quad \begin{array}{lll}
\texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \sqsubseteq c_1 \\
\texttt{[ ]}, x :: xs' & \rightarrow & \texttt{True} \sqsubseteq c_1 \\
y :: ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \sqsubseteq c_2 \\
y :: ys', x :: xs' & \rightarrow & \texttt{tt}
\end{array}
\end{array}$$

After the above rewrite we are left with the following property to be solved for c_1 and c_2:

$$\begin{array}{l}
\text{case } ys, xs \text{ of} \\
\quad \begin{array}{lll}
\texttt{[ ]}, \texttt{[ ]} & \rightarrow & \texttt{True} \sqsubseteq c_1 \\
\texttt{[ ]}, x :: xs' & \rightarrow & \texttt{True} \sqsubseteq c_1 \\
y :: ys', \texttt{[ ]} & \rightarrow & \texttt{null}\, ys' \sqsubseteq c_2 \\
y :: ys', x :: xs' & \rightarrow & \texttt{tt}
\end{array}
\end{array}$$

Using the solve function we discover $c_1 = \texttt{True}$, but cannot discover a value for c_2. However, as explained in Section 8.2, a partial solution is okay as long as we have a non-recursive data-type, as we can use the left-hand side of the initial approximation as a default value for any variables we could not solve. Fold discovery will therefore choose $c_2 = \texttt{le-len}\, ys\ (x :: xs)$.

Having discovered a potential substitution for c_1 and c_2, fold discovery checks that these values satisfy our rewritten property, which in this case they do. Since our property rewriting steps preserve $\Leftarrow$, we know that this substitution also satisfies our original inequality, so fold discovery has found

$$\texttt{le-len}\, ys\ (x :: xs) \sqsubseteq \text{fold}_{\texttt{Bool}} \langle \texttt{True}, \texttt{le-len}\, ys\ (x :: xs) \rangle\ (\texttt{le-len}\, ys\ xs)$$

In this chapter we have described the fold discovery technique, which helps fusion steps apply their fusion fact rewrite by discovering the definition of a fold function, invoking our theorem prover within this process. Now all the rewrite rules of Elea have been described, the next chapter will prove this rewrite system both sound and terminating.

Chapter 9

Termination and soundness

Prior chapters describe the term rewriting system $\xrightarrow{\mathfrak{R}}_+$ used by Elea. This chapter gives proofs that this relation is both terminating (Section 9.1 and sound (Section 9.2). By terminating I mean that this relation admits no infinite sequences of rewrites (Section 9.1). By sound I mean that, ignoring environment, if a rewrite $A \xrightarrow{\mathfrak{R}}_+ B$ is applicable then we have $A \mathrel{\mathfrak{R}} B$, where $\mathfrak{R}$ is either $\sqsubseteq$ or $\sqsupseteq$ (Section 9.2). Soundness is proven, in part, using a method I have called truncation fusion, a strengthening of fixed-point fusion.

9.1 Termination

This section is a proof that the rewrite system described in this thesis, $\xrightarrow{\mathfrak{R}}_+$, is terminating, viz. it admits no infinite sequence of rewrites. I prove this in Theorem 2 by showing that recursive uses of this rewrite rule are always smaller w.r.t. the ordering in Definition 9.1, which I prove to be well-founded in Lemma 9.2 using the lemma Lemma 9.1.

Definition 9.1 (Termination ordering for $\xrightarrow{\mathfrak{R}}_+$).

$$(\mathcal{H}, A) \rhd (\mathcal{H}', A') \quad \Leftrightarrow \quad \mathcal{H} \rhd \mathcal{H}' \lor (\mathcal{H} = \mathcal{H}' \land A' \in \mathsf{subterms}(A))$$

This is the well-founded ordering I will show proves every recursive usage of $\xrightarrow{\mathfrak{R}}_+$ obeys, where A is the term we are rewriting

---

Lemma 9.1 (Lexicographic composition preserves well-foundedness). If orders $>_A$ and $>_B$ are both well-founded, then the lexicographic composition of these orders is well-founded. This lexicographic composition $>_{A,B}$ is given by:

$$(a, b) >_{A,B} (a', b') \Leftrightarrow a >_A a' \lor (a = a' \land b >_B b')$$

Proof. Assume the contrary of well-foundedness, that an infinite chain $(a_1, b_1) >_{A,B} (a_2, b_2) >_{A,B}$ ... exists. From this, and the definition of $>_{A,B}$ we have that there is an infinite chain $a_1 \geq_A a_2 \geq_A$ .... Since $>_A$ is well-founded, there must exist an i s.t. $\forall (j > i) \,.\, a_i = a_j$. Therefore, we know there exists an infinite chain $b_i >_B b_{i+1} >_B$ ..., a contradiction since $>_B$ is well-founded. □

Lemma 9.2 (Elea's termination ordering is well-founded). $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$ from Definition 9.1 is well-founded.

Proof. We have that $\mathcal{H} \triangleright \mathcal{H}'$ on term histories is well-founded from Lemma 2.14 on page 42. The ordering $A > A' \Leftrightarrow A' \in \mathsf{subterms}(A)$ is trivially well-founded. As $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$ is the lexicographic composition of these two orderings, Lemma 9.1 gives us our proof. □

Lemma 9.3 ($\xrightarrow{\mathfrak{R}}$ calls $\xrightarrow{\mathfrak{R}}_+$ with decreasing arguments). For all terms A, B, A', B', type environments Γ and Γ', fact environments Φ and Φ', and history environments $\mathcal{H}$ and $\mathcal{H}'$.

$$\begin{array}{ll} \text{if} & \Gamma', \Phi', \mathcal{H}' \vdash A' \xrightarrow{\mathfrak{R}}_+ B' \text{ is an antecedent rewrite of } \Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \\ \text{then} & (\mathcal{H}, A) \triangleright (\mathcal{H}', A') \end{array}$$

This lemma is showing that every time $\xrightarrow{\mathfrak{R}}$ recursively calls $\xrightarrow{\mathfrak{R}}_+$, it does so with decreasing arguments.

Proof. By cases over all 47 rules which make up $\xrightarrow{\mathfrak{R}}$. Rather than enumerating every case I will divide them into three categories, and prove $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$ in every case. The first category are rewrite rules which have no antecedent rewrites, so this property trivially holds, e.g. rules 1 to 9. The second category apply antecedent rewrites using $\xrightarrow{\mathfrak{R}}_+$ on a sub-term while keeping $\mathcal{H}$ equal, e.g. rules 10 to 17. In this case we have $\mathcal{H}' = \mathcal{H}$ and $A' \in \mathsf{subterms}(A)$, hence $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$.

The third category are rules which call $\xrightarrow{\mathfrak{R}}_+$ on non-subterms, but increase the size of $\mathcal{H}$ by adding a term. In this case we have $|\mathcal{H}'| > |\mathcal{H}|$, and $\mathcal{H}' \notin \mathcal{W}_{\trianglelefteq}$, since this is a condition of $\xrightarrow{\mathfrak{R}}_+$ being applicable (Definition 4.3 on page 53), hence $\mathcal{H} \triangleright \mathcal{H}'$ and therefore $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$.

All 47 rules of $\xrightarrow{\mathfrak{R}}$ are in one of these three categories. □

Theorem 2 (Elea's rewrite rule $\xrightarrow{\mathfrak{R}}_+$ is terminating). Fix some terms A and B, type environment Γ, fact environment Φ, and history environment $\mathcal{H}$:

$$\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_+ B \text{ has a finite derivation}$$

Proof. By induction, assuming as an induction hypothesis:

$$\begin{array}{l} \forall \Gamma', \Phi', \mathcal{H}', A', B' \,.\, (\mathcal{H}, A) \triangleright (\mathcal{H}', A') \Rightarrow \\ \qquad \Gamma', \Phi', \mathcal{H}' \vdash A' \xrightarrow{\mathfrak{R}}_+ B' \text{ has a finite derivation.} \end{array}$$

The $\xrightarrow{\mathfrak{R}}_{+}$ rule (Definition 4.3 on page 53) has two cases. The non-transitive first case can be proven by showing:

$$\Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \text{ has a finite derivation}$$

The transitive second case can be proven by showing the above for the left antecedent, and using the induction hypothesis for the right antecedent. Hence, all that remains to be proven for both cases is the above property, which follows trivially from Lemma 9.3 and the induction hypothesis. □

9.2 Soundness

This entire section is a proof of Theorem 3, a property which represents the soundness of the rewriting system within my tool Elea. This proof relies on lemmas which are given throughout the rest of this section.

Definition 9.2 (Soundness of rewriting). These two definitions express that the $\xrightarrow{\mathfrak{R}}$ and $\xrightarrow{\mathfrak{R}}_{+}$ rewrite rules are sound, parameterised on the term being rewritten (A), and the history of previously rewritten terms ($\mathcal{H}$). These definitions are used to encode the inductive structure of my soundness proof.

$$\begin{array}{lcll}
\mathsf{Sound}(\mathcal{H}, A) & \overset{\text{def}}{\Leftrightarrow} & \text{forall} & \mathfrak{R}, \Gamma, \Phi, B \\
 & & \text{if} & \Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}} B \\
 & & \text{then} & \forall (\rho \in [\![\Gamma]\!]) \,.\, [\![\Phi]\!]\,(\rho) \Rightarrow [\![A]\!]\,\rho \;\mathfrak{R}\; [\![B]\!]\,\rho \\
 & & & \\
\mathsf{Sound}_{+}(\mathcal{H}, A) & \overset{\text{def}}{\Leftrightarrow} & \text{forall} & \mathfrak{R}, \Gamma, \Phi, B \\
 & & \text{if} & \Gamma, \Phi, \mathcal{H} \vdash A \xrightarrow{\mathfrak{R}}_{+} B \\
 & & \text{then} & \forall (\rho \in [\![\Gamma]\!]) \,.\, [\![\Phi]\!]\,(\rho) \Rightarrow [\![A]\!]\,\rho \;\mathfrak{R}\; [\![B]\!]\,\rho
\end{array}$$

Φ is a set of $\sqsubseteq$ or $\sqsupseteq$ relationships between terms, and $[\![\Phi]\!]\,(\rho)$ is a proof that all of these relationships hold in the term environment ρ (Definition 4.1 on page 52).

Lemma 9.4 (Lifting soundness of $\xrightarrow{\mathfrak{R}}$ to $\xrightarrow{\mathfrak{R}}_{+}$). Fix some term A and term history $\mathcal{H}$, then

$$\text{if} \quad \mathsf{Sound}(\mathcal{H}, A) \quad \text{then} \quad \mathsf{Sound}_{+}(\mathcal{H}, A)$$

Proof. This proof is by induction, assuming as an inductive hypothesis:

$$\forall (\mathcal{H}', A') \,.\, (\mathcal{H}, A) \rhd (\mathcal{H}', A') \Rightarrow \mathsf{Sound}_{+}(\mathcal{H}', A')$$

It remains to prove that that $\mathsf{Sound}_+(\mathcal{H}, A)$ holds for the two cases which define the $\xrightarrow{\mathfrak{R}}_+$ rule, given in Definition 4.3 on page 53. The first non-transitive case follows from $\mathsf{Sound}(\mathcal{H}, A)$. The second follows from $\mathsf{Sound}(\mathcal{H}, A)$, the transitivity of $\mathfrak{R}$, and $\mathsf{Sound}_+((\mathcal{H}, A), B)$ given $(\mathcal{H}, A) \notin \mathcal{W}_{\trianglelefteq}$. This latter property we can show using the induction hypothesis, as we have that $\forall B\ .\ (\mathcal{H}, A) \triangleright ((\mathcal{H}, A), B)$ if $(\mathcal{H}, A) \notin \mathcal{W}_{\trianglelefteq}$ from the definition of $\triangleright$. $\square$

Lemma 9.5 (Elea is sound). Fix some term A and term history $\mathcal{H}$, then

$$\begin{array}{ll} \text{if} & \forall(\mathcal{H}', A')\ .\ (\mathcal{H}, A) \triangleright (\mathcal{H}', A') \Rightarrow \mathsf{Sound}_+(\mathcal{H}', A') \quad \text{(hyp)} \\ \text{then} & \mathsf{Sound}(\mathcal{H}, A) \end{array}$$

Proof. By cases over all 47 rules which define $\xrightarrow{\mathfrak{R}}$. For all rules, we can always apply the hypothesis (hyp) to any antecedent uses of $\xrightarrow{\mathfrak{R}}_+$, since we always have $(\mathcal{H}, A) \triangleright (\mathcal{H}', A')$ from Lemma 9.3.

Rules 1 to 9, 18, 23, 31, 35, 36, 38 follow trivially from the denotation of νPCF terms. Rules 10 to 15, 19, 20, follow from the monotonicity of term denotations, and (hyp). Rule 17 follows from Lemma 9.7 and (hyp), and Rule 16 follows trivially from the soundness of Rule 17.

From (hyp) we trivially have the weaker property:

$$\forall(\mathcal{H}', A')\ .\ \mathcal{H} \triangleright \mathcal{H}' \Rightarrow \mathsf{Sound}_+(\mathcal{H}', A')$$

Using the above with Lemma 9.14 on page 131 gives us that our fusion rewrite rules which define $\mathcal{C} \oplus \mathcal{C}[\text{fix}\,(G)] \xrightarrow{\mathfrak{R}}$ are sound, from which it trivially follows that all fusion rewrites, Rules 24 to 30, are sound.

The soundness of the fission rewrites, Rules 32, 33, 34, and 37, follow from the lemmas given in Section 9.2.4 on page 133.

The theorem proving rewrite rules in Chapter 7 all follow from the denotation of terms, requiring (hyp) in the case of right transitivity, left transitivity, and traverse forall. The soundness of the least fixed-point rewrite (Rule 46) relies on the least fixed-point principle from Lemma 2.4 on page 32.

Finally, Rule 47 (fold discovery) is proven sound in Lemma 9.6, where (hyp) provides the first and second antecedents, and the encoding of Φ provides the third. $\square$

Theorem 3 (Elea is sound). For any term A, term history $\mathcal{H}$, and fact environment Φ, we have $\mathsf{Sound}_+(\mathcal{H}, A)$.

Proof. By induction, assuming as an induction hypothesis:

$$\forall(\mathcal{H}', A')\ .\ (\mathcal{H}, A) \triangleright (\mathcal{H}', A') \Rightarrow \mathsf{Sound}_+(\mathcal{H}', A')$$

From the induction hypothesis and Lemma 9.5, we have $\mathsf{Sound}(\mathcal{H}, A)$. From this and Lemma 9.4 we have our goal. $\square$

Lemma 9.6 (Fold discovery is sound). Fix some type environment Γ, and term environment $\rho \in [\![\Gamma]\!]$, variables h, $c_1...c_m$, and $x_1...x_n$, terms $A_1...A_n$, $C_1...C_m$, and G, term contexts $\mathcal{C}$ and $\mathcal{C}'$, and property term $\mathcal{Q}$.

$$\begin{array}{lll}
\text{let} & \mathcal{P} = \mathcal{C}[\text{fix}^a\,(G)] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle c_1, ..., c_m\rangle\,(\mathcal{C}'[\text{fix}^a\,(G)][A_1/x_1]...[A_n/x_n]) & \\
\text{if} & [\![\mathcal{P}]\!]\,\rho \sqsubseteq [\![\mathcal{Q}]\!]\,\rho & (1) \\
\text{and} & [\![\mathcal{Q}[C_1/c_1]...[C_m/c_m]]\!]\,\rho \sqsubseteq [\![\texttt{tt}]\!]\,\rho & (2) \\
\text{and} & [\![\text{fn } x_1, ..., x_n.\,\mathcal{C}'[\text{fix}\,(G)]]\!]\,\rho \sqsubseteq [\![h]\!]\,\rho & (3) \\
\text{then} & [\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\,(h\ A_1...A_n)]\!]\,\rho &
\end{array}$$

Proof. From (1), (2), transitivity, and monotonicity we have:

$$[\![\mathcal{P}[C_1/c_1]...[C_m/c_m]]\!]\,\rho \sqsubseteq [\![\texttt{tt}]\!]\,\rho$$

Inlining the definition of $\mathcal{P}$ gives:

$$[\![\mathcal{C}[\text{fix}^a\,(G)] \sqsubseteq \text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\,(\mathcal{C}'[\text{fix}^a\,(G)][A_1/x_1]...[A_n/x_n])]\!]\,\rho \sqsubseteq [\![\texttt{tt}]\!]\,\rho$$

From the above and Lemma 7.2 on page 101 we have:

$$[\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fold}_{\boldsymbol{T}}\langle C_1, ..., C_m\rangle\,(\mathcal{C}'[\text{fix}^a\,(G)][A_1/x_1]...[A_n/x_n])]\!]\,\rho$$

Applying (3) to the above using monotonicity gives us our goal. □

9.2.1 Traversing into pattern match branches is sound

This section focuses on proving the soundness of Rule 17 in Lemma 9.9, using Lemma 9.7 for the bulk of the proof.

Lemma 9.7 (Pattern matches can be used as equations). Let Γ be a type environment, $\rho \in [\![\Gamma]\!]$ a term environment, (case M of $P_1 \to A_1$ ... $P_n \to A_n$) and B be terms well-typed by Γ, $i \in [1..n]$, and $\mathfrak{R} \in \{\sqsubseteq, \sqsupseteq\}$. $\Gamma \lhd P_i$ denotes the type environment Γ extended with the variables matched in the pattern P_i, given in Definition 2.8 on page 21.

$$\begin{array}{ll}
\text{if} & \forall(\rho' \in [\![\Gamma \lhd P_i]\!])\,.\,\rho \le \rho' \wedge [\![M = P_i]\!]\,\rho' \Rightarrow [\![A_i]\!]\,\rho'\ \mathfrak{R}\ [\![B]\!]\,\rho' \\
\text{then} & [\![\text{case } M \text{ of } ...\ P_i \to A_i\ ...]\!]\,\rho\ \mathfrak{R}\ [\![\text{case } M \text{ of } ...\ P_i \to B\ ...]\!]\,\rho
\end{array}$$

Proof. We first case-split on whether there exists some j and $d_1...d_n$ such that $[\![M]\!]\,\rho = \mathsf{inj}_j\,(d_1, ..., d_n)$ and $\mathsf{findPattern}_j(P_1...) = P_i$, viz. whether we are pattern matching to the P_i branch. If this doesn't hold, then our property trivially holds, as only the P_i branch has changed between the terms. It remains the prove that the property holds when there exists some j and $d_1...d_n$ such that $[\![M]\!]\,\rho = \mathsf{inj}_j\,(d_1, ..., d_n)$ and $\mathsf{findPattern}_j(P_1...) = P_i$. Since

$\mathsf{findPattern}_j(P_1 ...) = P_i$ we know there exists some $x_1...x_n$ such that $\text{con}_{\boldsymbol{T}}\langle j\rangle\ x_1...x_n = P_i$. Let $\rho' \in [\![\Gamma \lhd\ P_i]\!] = \rho[x_1 \mapsto d_1]...[x_n \mapsto d_n]$, we can now show that $[\![M]\!]\,\rho' = [\![P_i]\!]\,\rho'$:

$$\begin{aligned}
&[\![P_i]\!]\,\rho' \\
&\qquad = \{ \text{ by } \mathsf{findPattern}_j(P_1 ...) = P_i \ \} \\
&[\![\text{con}_{\boldsymbol{T}}\langle j\rangle\ x_1...x_n]\!]\,\rho' \\
&\qquad = \{ \text{ by denotation of terms } \} \\
&\mathsf{inj}_j\ ([\![x_1]\!]\,\rho', ..., [\![x_n]\!]\,\rho') \\
&\qquad = \{ \text{ by denotation of variables } \} \\
&\mathsf{inj}_j\ (d_1, ..., d_n)
\end{aligned}$$

Using this, we can invoke the antecedent to the property we are proving, since trivially $\rho \leq \rho'$, yielding $[\![A_i]\!]\,\rho'\ \mathfrak{R}\ [\![B]\!]\,\rho'$, which allows us to complete this proof:

$$\begin{aligned}
&\Big[\!\!\Big[\text{case } M \text{ of } ...\ P_i \to A_i\ ...\Big]\!\!\Big]\,\rho \\
&\qquad = \{ \text{ by denotation of pattern matches (Definition 2.39 on page 36) } \} \\
&[\![A_i]\!]\,\rho' \\
&\qquad \mathfrak{R}\ \{ \text{ by } [\![A_i]\!]\,\rho' \sqsubseteq [\![B]\!]\,\rho' \ \} \\
&[\![B]\!]\,\rho' \\
&\qquad = \{ \text{ by denotation of pattern matches } \} \\
&\Big[\!\!\Big[\text{case } M \text{ of } ...\ P_i \to B\ ...\Big]\!\!\Big]\,\rho
\end{aligned}$$

□

Definition 9.3 (Extending type environments). Given two type environments Γ and Γ', Γ is a sub-environment of Γ', denoted $\Gamma \leq \Gamma'$, iff $\forall(x \in \mathsf{dom}\,(\Gamma))\ .\ \Gamma(x) = \Gamma'(x)$.

Definition 9.4 (Extending term environments). Given some type environments $\Gamma \leq \Gamma'$, a term environment $\rho \in [\![\Gamma]\!]$ is a sub-environment of $\rho' \in [\![\Gamma']\!]$, denoted $\rho \leq \rho'$ iff $\forall(x \in \mathsf{dom}\,(\rho))\ .\ \rho(x) = \rho'(x)$.

Lemma 9.8 ($[\![\Phi]\!]$ is monotonic). Fix some term environments ρ and ρ', and fact environment Φ.

$$\rho \leq \rho' \Rightarrow ([\![\Phi]\!]\,\rho \Rightarrow [\![\Phi]\!]\,\rho')$$

Proof. $[\![\Phi]\!]\,\rho$ is only defined if all free variables in Φ are bound to values in ρ, hence the extra bindings in ρ' will not affect the value of $[\![\Phi]\!]\,\rho'$. □

Lemma 9.9 (Rule 17 is sound).

$$\begin{array}{ll} \text{if} & \forall(\rho' \in [\![\Gamma \lhd\ P_i]\!]) \,.\, [\![\Phi, M \sqsubseteq P_i, P_i \sqsubseteq M]\!]\,\rho' \Rightarrow [\![A_i]\!]\,(\rho')\;\mathfrak{R}\;[\![A'_i]\!]\,\rho' \\ \text{then} & \forall(\rho \in \Gamma) \,.\, [\![\Phi]\!]\,\rho \Rightarrow \\ & \qquad [\![\text{case } M \text{ of } ...\ P_i \to A_i\ ...]\!]\,\rho\;\mathfrak{R}\;[\![\text{case } M \text{ of } ...\ P_i \to A'_i\ ...]\!]\,\rho \end{array}$$

Proof. Fix some $\rho \in \Gamma$ and assume $[\![\Phi]\!]\,\rho$. It remains to prove:

$$\begin{array}{l} [\![\text{case } M \text{ of } ...\ P_i \to A_i\ ...]\!]\,\rho\;\mathfrak{R}\;[\![\text{case } M \text{ of } ...\ P_i \to A'_i\ ...]\!]\,\rho \\ \qquad \Leftarrow \{ \text{ by Lemma 9.7 } \} \\ \forall(\rho' \in [\![\Gamma \lhd\ P_i]\!]) \,.\, \rho \le \rho' \wedge [\![M = P_i]\!]\,\rho' \Rightarrow [\![A_i]\!]\,\rho'\;\mathfrak{R}\;[\![B]\!]\,\rho' \\ \qquad \Leftarrow \{ \text{ by Lemma 9.8 } \} \\ \forall(\rho' \in [\![\Gamma \lhd\ P_i]\!]) \,.\, [\![\Phi]\!]\,\rho' \wedge [\![M = P_i]\!]\,\rho' \Rightarrow [\![A_i]\!]\,\rho'\;\mathfrak{R}\;[\![B]\!]\,\rho' \\ \qquad \Leftrightarrow \{ \text{ by Definition 4.1 on page 52 } \} \\ \forall(\rho' \in [\![\Gamma \lhd\ P_i]\!]) \,.\, [\![\Phi, M \sqsubseteq P_i, P_i \sqsubseteq M]\!]\,\rho' \Rightarrow [\![A_i]\!]\,\rho'\;\mathfrak{R}\;[\![B]\!]\,\rho' \\ \qquad \Leftrightarrow \{ \text{ by assumption } \} \\ \qquad \top \end{array}$$

□

9.2.2 Properties of truncated fixed-points

This section gives lemmas relating to truncated fixed-points which are used in the soundness proofs for my fusion and fission rules.

Lemma 9.10. Given a term F and term context $\mathcal{C}$, $[\![\mathcal{C}[\text{fix}^0\,(F)]]\!] \sqsubseteq [\![\mathcal{C}[\text{fix}^1\,(F)]]\!] \sqsubseteq ...$ is a chain whose least upper-bound is $[\![\mathcal{C}[\text{fix}\,(F)]]\!]$.

Proof. By the denotation of truncated and least fixed-points, and that term contexts are continuous. □

Lemma 9.11 (Truncation fusion). Given a domain $(D, \sqsubseteq)$, chain $d_0 \sqsubseteq d_1 \sqsubseteq ...$ in D, and continuous function $h : D \to D$

$$\begin{array}{lll} \text{if} & d_0 = \bot & (1) \\ \text{and} & \forall(i \in \mathbb{N}) \,.\, d_{i+1} \sqsubseteq h\ d_i & (2) \\ \text{then} & \bigsqcup_i d_i \sqsubseteq \mathsf{fix}\,(h) & \end{array}$$

Proof. We can show $\forall(i \in \mathbb{N}) \,.\, d_i \sqsubseteq \mathsf{fix}\,(h)$ by induction on i with (1) the base case and (2) the inductive case. Hence $\mathsf{fix}\,(h)$ is an upper-bound of the chain d_i and hence it is more defined or equal to the least upper-bound. □

This lemma also holds for $=$ in place of $\sqsubseteq$.

Lemma 9.12 (Pre-truncation rule). Let G and H be terms, $\mathcal{C}$ a term context, Γ a type environment, $\rho \in [\![\Gamma]\!]$ a value environment.

if	$\forall (b \in \mathbb{N})\ .\ \left[\!\left[H\, \mathcal{C}[\mathrm{fix}^{b}\,(G)]\right]\!\right] \rho \sqsubseteq \left[\!\left[\mathcal{C}[\mathrm{fix}^{b+1}\,(G)]\right]\!\right] \rho$	(1)
then	$\forall (a \in \mathbb{N})\ .\ [\![\mathrm{fix}^{a}\,(H)]\!] \rho \sqsubseteq [\![\mathcal{C}[\mathrm{fix}^{a}\,(G)]]\!] \rho$	
and	$[\![\mathrm{fix}\,(H)]\!] \rho \sqsubseteq [\![\mathcal{C}[\mathrm{fix}\,(G)]]\!] \rho$	

Proof. I only prove the first goal, as the second follows from this goal and Lemma 2.2 on page 29. To make this proof clearer I will drop the $[\![...]\!]\, \rho$ around every term. This proof is done by induction on a. The base case below holds trivially.

$$\bot \sqsubseteq \mathcal{C}[\bot]$$

For the inductive case we assume

$$\mathrm{fix}^{a}\,(H) \sqsubseteq \mathcal{C}[\mathrm{fix}^{a}\,(G)] \qquad (2)$$

And show

$\mathrm{fix}^{a+1}\,(H)$

$\quad = \{$ by denotation of truncated fixed-points $\}$

$H\,(\mathrm{fix}^{a}\,(H))$

$\quad \sqsubseteq \{$ by inductive assumption (2) and monotonicity $\}$

$H\,\mathcal{C}[\mathrm{fix}^{a}\,(G)]$

$\quad \sqsubseteq \{$ by (1) $\}$

$\mathcal{C}[G\,(\mathrm{fix}^{a}\,(G))]$

$\quad = \{$ by denotation of truncated fixed-points $\}$

$\mathcal{C}[\mathrm{fix}^{a+1}\,(G)]$

□

Lemma 9.13 (Post-truncation rule). Let G and H be terms, $\mathcal{C}$ a term context, Γ a type environment, $\rho \in [\![\Gamma]\!]$, and $a \in \mathbb{N}$.

if	$\forall (b \in \mathbb{N})\ .\ \left[\!\left[\mathcal{C}[\mathrm{fix}^{b+1}\,(G)]\right]\!\right] \rho \sqsubseteq \left[\!\left[H\, \mathcal{C}[\mathrm{fix}^{b}\,(G)]\right]\!\right] \rho$	(1)
and	$[\![\mathcal{C}[\bot]]\!] \rho = \bot$	(2)
then	$\forall (a \in \mathbb{N})\ .\ [\![\mathcal{C}[\mathrm{fix}^{a}\,(G)]]\!] \rho \sqsubseteq [\![\mathrm{fix}^{a}\,(H)]\!] \rho$	
and	$[\![\mathcal{C}[\mathrm{fix}\,(G)]]\!] \rho \sqsubseteq [\![\mathrm{fix}\,(H)]\!] \rho$	

Proof. This lemma can be shown using the same proof as for Lemma 9.12 with all instances of $\sqsubseteq$ replaced by $\sqsupseteq$. The extra antecedent (2) is for the base case of the proof by induction. □

9.2.3 Soundness of fusion

This section proves the soundness of the fusion rules from Chapter 5, where what it means for fusion to be sound is established in Definition 9.5. This property is then proven in Lemma 9.14 under the condition that its antecedent $\xrightarrow{\mathfrak{R}}_+$ rewrite is sound, using Lemma 9.16 to prove the ω-fusion rule sound, and Lemma 9.15 to prove the truncation fusion rewrite sound.

Definition 9.5 (Soundness of fusion rewrites). The property $\mathsf{Sound}_\oplus(\mathcal{H})$ expresses that all fusion rewrites with term history $\mathcal{H}$ preserve the relation $\mathfrak{R}$.

$$\begin{array}{lll} \mathsf{Sound}_\oplus(\mathcal{H}) & \overset{\text{def}}{\Leftrightarrow} & \text{forall} \quad \mathfrak{R}, \Gamma, \Phi, \mathcal{C}, G, B \\ & & \text{if} \quad \Gamma, \Phi, \mathcal{H} \vdash \mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} B \\ & & \text{then} \quad \forall(\rho \in [\![\Gamma]\!]) \,.\, [\![\Phi]\!]\,(\rho) \Rightarrow [\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho \,\mathfrak{R}\, [\![B]\!]\,\rho \end{array}$$

Lemma 9.14 (Conditional soundness of fusion rewrite rules). This lemma asserts that both fusion rewrite rules are sound, given that their antecedent rewrite is sound.

Fix some context $\mathcal{C}$, term G.

$$\begin{array}{ll} \text{if} & \forall(\mathcal{H}', A) \,.\, \mathcal{H} \rhd \mathcal{H}' \Rightarrow \mathsf{Sound}_+(\mathcal{H}', A) \\ \text{then} & \mathsf{Sound}_\oplus(\mathcal{H}) \end{array}$$

Proof. There are two rules which define the fusion rewrite $\mathcal{C} \oplus \text{fix}\,(G) \xrightarrow{\mathfrak{R}} A$, and hence two rules we must prove this property for. These are truncation fusion (Rule 22 on page 68) and ω fusion (Rule 21 on page 67). The prove of the above property for truncation fusion is Lemma 9.15, and the proof for ω fusion is Lemma 9.16. In both cases we invoke the assumption in order to get that the antecedent rewrite to the fusion step is sound, viz. $\mathsf{Sound}_+(\mathcal{H}', A)$. We always have that $\mathcal{H} \rhd \mathcal{H}'$, as both fusion rewrites extend their term history. □

Lemma 9.15 (Soundness of truncation fusion). This lemma expresses the soundness of the truncation fusion rewrite rule.

Let G and H be terms, $\mathcal{C}$ a term context, h be fresh variables, Γ a type environment, $\rho \in [\![\Gamma]\!]$ a value environment and τ a type such that $\Gamma \vdash \mathcal{C}[\text{fix}\,(G)] : \tau$:

$$\begin{array}{lll} \text{if} & [\![\Phi]\!]\,\rho & (1) \\ \text{and} & [\![\mathcal{C}[\bot]]\!] = \bot & (2) \\ \text{and} & \forall(\eta \in [\![\Gamma[h \mapsto \tau]]\!]\,, a \in \mathbb{N}) \,. & \\ & \quad [\![\Phi]\!]\,\eta \wedge [\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\eta \sqsubseteq [\![h]\!]\,\eta \Rightarrow [\![\mathcal{C}[G\,(\text{fix}^a\,(G))]]\!]\,\eta \sqsubseteq [\![H]\!]\,\eta & (3) \\ \text{then} & [\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fix}\,(\text{fn}\ h.\ H)]\!]\,\rho & \end{array}$$

Our assumption (3) expresses the inductive hypothesis to our overall proof, viz. that the antecedent $\xrightarrow{\sqsubseteq}_+$ rewrite within fixed-point fusion is sound.

Proof. Fix some $a \in \mathbb{N}$ and let $\eta = \rho[h \mapsto [\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho]$. Since h is not free in Φ, as it was a fresh variable, from (1) we know $[\![\Phi]\!]\,\eta$. Applying this to (2) gives us

$$[\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\eta \sqsubseteq [\![h]\!]\,\eta \Rightarrow [\![\mathcal{C}[\text{fix}^{a+1}\,(G)]]\!]\,\eta \sqsubseteq [\![H]\!]\,\eta$$

We can beta-abstract h in H in the consequent of the above, yielding

$$[\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\eta \sqsubseteq [\![h]\!]\,\eta \Rightarrow [\![\mathcal{C}[\text{fix}^{a+1}\,(G)]]\!]\,\eta \sqsubseteq [\![\text{fn}\ h.\ H]\!]\,\eta\,([\![h]\!]\,\eta)$$

As h was a fresh variable it is not free in $\mathcal{C}$ or G, so its value in η does not change their denotations, hence we can replace η with ρ on the left-hand side of both antecedent and consequent in the above. We can also replace η with ρ in $[\![\text{fn}\ h.\ H]\!]\,\eta$ for the same reason.

$$[\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho \sqsubseteq [\![h]\!]\,\eta \Rightarrow [\![\mathcal{C}[\text{fix}^{a+1}\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fn}\ h.\ H]\!]\,\rho\,([\![h]\!]\,\eta)$$

Applying the mapping of h in η to the above gives us

$$[\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho \sqsubseteq [\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho \Rightarrow [\![\mathcal{C}[\text{fix}^{a+1}\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fn}\ h.\ H]\!]\,\rho\,([\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho)$$

The antecedent holds by reflexivity, yielding

$$[\![\mathcal{C}[\text{fix}^{a+1}\,(G)]]\!]\,\rho \sqsubseteq [\![\text{fn}\ h.\ H]\!]\,\rho\,([\![\mathcal{C}[\text{fix}^a\,(G)]]\!]\,\rho)$$

The above, with (2), allows us to invoke Lemma 9.13 to get our goal □

Lemma 9.16 (Soundness of ω-fusion). This lemma expresses the soundness of the ω-fusion rewrite rule.

Let G and H be terms, $\mathcal{C}$ a term context, h be fresh variables, Γ a type environment, $\rho \in [\![\Gamma]\!]$ a value environment and τ a type such that $\Gamma \vdash \mathcal{C}[\text{fix}\,(G)] : \tau$:

$$\begin{array}{lll} \text{if} & [\![\Phi]\!]\,\rho & (1) \\ \text{and} & \forall(\eta \in [\![\Gamma[h \mapsto \tau]]\!])\ . & \\ & \quad [\![\Phi]\!]\,\eta \wedge [\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\eta \sqsupseteq [\![h]\!]\,\eta \Rightarrow [\![\mathcal{C}[G\ (\text{fix}\,(G))]]\!]\,\eta \sqsupseteq [\![H]\!]\,\eta & (2) \\ \text{then} & [\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho \sqsupseteq [\![\text{fix}\,(\text{fn}\ h.\ H)]\!]\,\rho & \end{array}$$

Our assumption (2) expresses the inductive hypothesis to our overall proof, viz. that the antecedent $\xrightarrow{\sqsubseteq}_+$ rewrite within fixed-point fusion is sound.

Proof. Following the same process as in the proof of the previous lemma, but exchanging $\sqsubseteq$ for $\sqsupseteq$ and $\text{fix}^a\,(G)$ for $\text{fix}\,(G)$, we can derive

$$[\![\mathcal{C}[G\ (\text{fix}\,(G))]]\!]\,\rho \sqsupseteq [\![(\text{fn}\ h.\ H)\ \mathcal{C}[G\ (\text{fix}\,(G))]]\!]\,\rho$$

By the denotation of term application, and that $[\![\text{fix}\,(G)]\!] = [\![G\ (\text{fix}\,(G))]\!]$, the above can be rewritten to

$$[\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho \sqsupseteq [\![\text{fn}\ h.\ H]\!]\,\rho([\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho)$$

This expresses that $[\![\mathcal{C}[\text{fix}\,(G)]]\!]\,\rho$ is a pre-fixed-point of $[\![\text{fn}\ h.\ H]\!]\,\rho$, and so by the least fixed-point principle (Lemma 2.4 on page 32) we have our goal. □

9.2.4 Soundness of fission

This section proves the soundness of the fission rewrite rules in Chapter 6. Recall that all four of these rules have the shape

$$\frac{\Gamma[g \mapsto \tau], \Phi, (\mathcal{H}, \mathrm{fix}^a\,(H)) \vdash H\,\mathcal{C}[g] \xrightarrow{\mathfrak{R}}_{+} \mathcal{C}[G]}{\Gamma, \Phi, \mathcal{H} \vdash \mathrm{fix}^a\,(H) \xrightarrow{\mathfrak{R}} \mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]}$$

To prove these rules sound I prove the soundness of the generalised rule above, of which our fission rules are specific instances. Lemma 9.18 proves this for $\mathfrak{R} = \sqsubseteq$ and Lemma 9.17 proves it for $\mathfrak{R} = \sqsupseteq$. When $\mathfrak{R} = \sqsupseteq$ we have the added requirement that $[\![\mathcal{C}[\bot]]\!] = \bot$. This $\sqsupseteq$ case is only used by accumulation fission, and the context used in that rule is trivially strict, since the hole occurs as a topmost function call. All fission rules are defined for both truncated and least fixed-points, hence why each soundness lemma has two consequents.

Lemma 9.17 ($\sqsupseteq$ fission). Given terms G and H, term context $\mathcal{C}$, type environment Γ, value environment $\rho \in [\![\Gamma]\!]$, $a \in \mathbb{N}$, fresh variable g, and type τ s.t. $\Gamma \vdash [\![\mathrm{fix}^a\,(G)]\!] : \tau$:

$$\begin{array}{lll}
\text{if} & [\![\Phi]\!]\,\rho & (1)\\
\text{and} & \forall(\eta \in [\![\Gamma[g \mapsto \tau]]\!])\,.\ [\![\Phi]\!]\,\eta \Rightarrow [\![H\,\mathcal{C}[g]]\!]\,\eta \sqsupseteq [\![\mathcal{C}[G]]\!]\,\eta & (2)\\
\text{and} & [\![\mathcal{C}[\bot]]\!] = \bot & (3)\\
\text{then} & [\![\mathrm{fix}^a\,(H)]\!]\,\rho \sqsupseteq [\![\mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho & \\
\text{and} & [\![\mathrm{fix}\,(H)]\!]\,\rho \sqsupseteq [\![\mathcal{C}[\mathrm{fix}\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho &
\end{array}$$

Our assumption (2) expresses the inductive hypothesis to our overall proof, viz. that the antecedent $\xrightarrow{\sqsubseteq}_{+}$ rewrite within fixed-point fission is sound.

Proof. This proof proceeds in the same way as that of Lemma 9.15, in that I show the assumptions allow us to apply Lemma 9.13, which in this case directly proves the goal.

Let $\eta = \rho[g \mapsto [\![\mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho]$, since g is not free in Φ, as it is a fresh variable, from (1) we have $[\![\Phi]\!]\,\eta$. Applying this to (2) gives us

$$[\![H\,\mathcal{C}[g]]\!]\,\eta \sqsubseteq [\![\mathcal{C}[G]]\!]\,\eta$$

We can beta abstract g in G

$$[\![H\,\mathcal{C}[g]]\!]\,\eta \sqsubseteq [\![\mathcal{C}[(\mathrm{fn}\ g.\,G)\ g]]\!]\,\eta$$

Now we apply the mapping of g in η to get

$$[\![H\,\mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]]\!]\,\eta \sqsubseteq [\![\mathcal{C}[(\mathrm{fn}\ g.\,G)\ (\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G))]]\!]\,\eta$$

We can use the fact that g will not be free in any of the above terms to replace the ηs with ρs. That, and applying the denotation of truncated fixed-points, gives us

$$[\![H\,\mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho \sqsubseteq [\![\mathcal{C}[\mathrm{fix}^{a+1}\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho$$

Using the above and (3) with Lemma 9.13 gives us our goal. □

Lemma 9.18 ($\sqsubseteq$ truncated fission). Given terms G and H, term context $\mathcal{C}$, type environment Γ, value environment $\rho \in [\![\Gamma]\!]$, $a \in \mathbb{N}$, fresh variable g, and type τ s.t. $\Gamma \vdash [\![\mathrm{fix}^a\,(G)]\!] : \tau$:

$$\begin{array}{lll}
\text{if} & [\![\Phi]\!]\,\rho & (1) \\
 & \forall(\eta \in [\![\Gamma[g \mapsto \tau]]\!]) \,.\, [\![\Phi]\!]\,\eta \Rightarrow [\![H\,\mathcal{C}[g]]\!]\,\eta \sqsubseteq [\![\mathcal{C}[G]]\!]\,\eta & (2) \\
\text{then} & [\![\mathrm{fix}^a\,(H)]\!]\,\rho \sqsubseteq [\![\mathcal{C}[\mathrm{fix}^a\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho & \\
\text{and} & [\![\mathrm{fix}\,(H)]\!]\,\rho \sqsubseteq [\![\mathcal{C}[\mathrm{fix}\,(\mathrm{fn}\ g.\,G)]]\!]\,\rho &
\end{array}$$

Our assumption (2) expresses the inductive hypothesis to our overall proof, viz. that the antecedent $\xrightarrow{\sqsubseteq}_+$ rewrite within fixed-point fission is sound.

Proof. We can prove this property using the same method as in Lemma 9.17, but with $\sqsupseteq$ replaced with $\sqsubseteq$, and Lemma 9.12 in place of Lemma 9.13, hence why we no longer need the strictness of $\mathcal{C}$ as an assumption. $\square$

Chapter 10

Evaluating Elea

This thesis describes the Elea theorem prover, a tool able to prove approximations between terms in a pure, call-by-name language with non-strict data-types. This section compares Elea with four existing tools that have similar domains: HOSC, HipSpec, Zeno, and Oyster/Clam, using an established set of properties from the literature.

Section 10.1 describes the set of properties which will be given to each of the five tools to see which can prove which[1]. Section 10.2 details each of the four tools Elea will be compared to, how they are designed and how their domain of properties differs from Elea's. The table of which tools can prove which properties is given in Section 10.3, and these results are analysed in Section 10.4.

The final section of this chapter, Section 10.5, discusses a potential extension which may allow Elea to recover some of the properties from this test set it was unable to prove. This method is an extension of the Elea's fission rules such that they would preserve equivalence, by using the seq syntax.

10.1 Test properties

This section explains the choice of properties used in this chapter to compare the theorem proving capabilities of Elea to that of other tools. All but 8 of the 143 properties used were taken from the literature on automated proof by induction [18]. The remaining 8 I chose myself as properties which cannot be proved by induction.

Throughout this thesis I have extolled the necessity of proving approximation properties, rather than equivalences, when dealing with non-total terms and non-strict data-types. Indeed, of the 135 equivalences taken from the induction literature, only 33 still hold if terms are allowed to contain undefinedness. The automated induction provers these properties have been used to test all assume that input terms are totally defined, but Elea does not.

If Elea was only able to prove equivalence, it would have just 40 properties to test in this chapter, as one of the non-induction properties only holds as an approximation, and

[1]The source code for Elea, along with the definitions of all the functions used in this chapter, which are not featured in Appendix A, can be found at `http://github.com/wsonnex/elea/`.

102 of the 135 equivalences from the induction literature do not hold for non-total terms. However, of these 102 properties, all but 17 can be made to hold by converting them into approximations. For example, `add` $x\,x \equiv$ `double` x does not hold for non-total terms, but `add` $x\,x \sqsubseteq$ `double` x does. The remaining 17 properties do not hold as approximations in either direction, and so are not applicable to Elea, but I have left them in the test set as it makes for an easier comparison with the induction literature, and marked them as n/a for Elea.

10.2 Compared tools

The previous section motivated the properties chosen to compare Elea to other theorem provers. This section describes each of the four provers Elea is compared to: HOSC, HipSpec, Zeno, and Oyster/Clam.

10.2.1 HOSC

HOSC [40, 39] is the most similar tool to Elea, in both domain and operation. It proves equivalence properties between terms in an input language isomorphic to Elea's, by supercompiling both terms and then checking for α-equality. Like Elea, it uses the homeomorphic embedding [41], but a more advanced version, to ensure termination. As with other supercompilers, and unlike Elea, it uses the most-specific generalisation heuristic to generalise terms such that they can be supercompiled [39].

Unlike HipSpec and Zeno, Elea and HOSC do not assume their input terms are total. As discussed in my introductory chapter and the previous section, many equivalences fail to hold when terms can contain undefinedness, and hence many of the properties tested do not hold when we assume totality. Due to this, 85 of the 135 properties from the induction literature have been converted into approximations where they did not hold as equivalences. Since HOSC cannot prove approximation properties, these are outside its domain, and have been marked "n/a" in the table of results.

HOSC also has not been designed to allow preconditions on the inputs to an equivalence, such as in the property Z17: `assert True <- le n 0 in n` $\equiv$ `0`. Although I could have encoded these properties as an equivalence, such as `if le n 0 then n else m` $\equiv$ `if le n 0 then 0 else m`, I found that HOSC was unable to prove any of the properties encoded this way, and so I have also marked these properties as "n/a", as HOSC has clearly not been designed with this misuse in mind.

10.2.2 HipSpec

HipSpec [18] is the tool most dissimilar to Elea in design. It is a bottom-up induction prover which uses randomised testing to generate a background theory of equivalences between terms formed from a given set of functions [19]. HipSpec then invokes an external theorem prover, such as an SMT solver, providing this set of generated equivalences, along with an induction schema, in order to prove the property. As with the Zeno prover it assumes all terms are total, and so any properties expressed as an approximation are given to HipSpec as an equivalence.

The method by which it generates this background theory is as follows. Starting with the given set of functions, HipSpec generates the set of well-typed terms which can be formed from these functions, up to a given maximum number of function applications and variables. Using the QuickCheck framework [17] to generate random input values, it is able to group these terms into those which give equal values for all equal random inputs. The upshot of this is that we end up with sets of terms which cannot be proved unequal using random testing, which means that all the terms in these sets are very likely to be equivalent. All these potential term equivalences can then be fed to an external theorem prover, and those which are provable can then be used as a background theory when proving our goal property. Equivalences which are less general than others are discovered using a subsumption checker, and discarded.

This method is experimentally very effective, and completely sidesteps the problem of term generalisation. That is to say, HipSpec has no need of a heuristic to decide how to generalise a property, since it builds its properties up by combinating terms, rather than by generalising subgoals. Its subsumption checker also removes equivalences which are less general than others it has found, so rather than guessing a generalisation, it simply has to check if one property generalises another - a much easier problem.

A current weakness of HipSpec is that it lacks the ability to conjecture lemmas involving implication, and so cannot prove any properties which require such a lemma. For example, `sorted (isort xs)` $\sqsubseteq$ `True` requires `assert True <- sorted xs in sorted (insert x xs)` $\sqsubseteq$ `True`, which is why HipSpec cannot prove this property. However, there is current research into automating the discovery of these conditional lemmas in HipSpec [70].

10.2.3 Zeno

Zeno [64] is a top-down automated induction prover for equivalence properties, a tool which I myself co-developed with Sophia Drossopoulou and Susan Eisenbach. Guided by the shape of the functions on either side of the equivalence, it successively applies tactics such as generalisation and induction in order to reduce properties to hopefully simpler subgoals. The heuristics which guide its usage of generalisation are very similar to the most-specific generalisation technique of supercompilation, except that most-specific generalisation considers the entire shape of a term, whereas Zeno considers only the shape of its pattern matches. As with the HipSpec tool, but unlike HOSC and Elea, Zeno assumes all terms are total, and so any properties expressed as an approximation are given to Zeno as an equivalence.

Elea is in many ways the spiritual successor to Zeno. Zeno gradually unrolls functions, building up an induction schema as it goes. Elea's fusion method can be thought of as considering the functions to be their own induction schemas, and, in applying fusion to combine nested function calls, it is also building up an overall induction schema for the proof by unrolling functions. Zeno tries to unroll function calls within a term in the correct order, building up an induction schema which can complete the proof. Elea tries to fuse a term into one large function, viz. a term in fixed-point promoted form, such that the proof can be completed by just one unrolling of this single function, viz. by the least fixed-point principle.

10.2.4 Oyster/Clam

Oyster/Clam is another top-down automated induction prover developed by Bundy et al. [14], which uses a technique called rippling [12] to guide its application of function unfolding and any lemmas which have been supplied. When it fails to apply an induction hypothesis and the proof process becomes blocked, it uses a number of techniques referred to as "proof critics" to discover a step the prover should have taken earlier in the proof process, or a lemma which can unblock the proof.

There are two proof critics of particular relevance to Elea, "lemma calculation" and "lemma speculation", which I will now detail. However, before going futher, I would like to note that the second set of properties in this test suite, P1 to P50, were taken from the original test set used to evaluate Oyster/Clam. Unfortunately, I was unable to acquire results for the other two sets of properties for this prover, so only these results are presented.

10.2.4.1 Lemma calculation

The lemma calculation critic unblocks the proof of an equation by conjecturing a lemma which is the generalisation of a common sub-term on either side of the property. For example, let's say we were trying to prove the property $\texttt{rev}\,(\texttt{rev}\,xs) = xs$ by induction on xs, Oyster/Clam will become blocked at trying to prove:

$$\texttt{rev}\,(\texttt{app}\,(\texttt{rev}\,xs)\,\texttt{[}x\texttt{]}) = x\ \texttt{::}\ \texttt{rev}\,(\texttt{rev}\,xs)$$

The lemma calculation technique will generalise the occurrence of $\texttt{rev}\,xs$ to some new ys on either side of the above, yielding the lemma below, which can then be proved separately. Using this lemma will allow Oyster/Clam to complete its proof of $\texttt{rev}\,(\texttt{rev}\,xs) = xs$.

$$\texttt{rev}\,(\texttt{app}\,ys\ \texttt{[}x\texttt{]}) = x\ \texttt{::}\ \texttt{rev}\,ys$$

In comparison, Elea proves the property $\texttt{rev}\,(\texttt{rev}\,xs) = xs$ by rewriting $\texttt{rev}\,(\texttt{rev}\,xs) \xrightarrow{\sqsubseteq}_{+} xs$, independently of the overall property - see Example 5.4 on page 73. The generalisation step equivalent to the one the lemma calculation critic performs occurs implicitly within Elea's rewrite process when it applies fix-fix fusion, which generalises all arguments to the inner fixed-point and, in this case, generalises $\texttt{rev}\,xs$.

10.2.4.2 Lemma speculation and middle-out reasoning

Lemma speculation occurs when Oyster/Clam is unable to apply an induction hypothesis as a rewrite. It will attempt to speculate a lemma based on how it expects the induction hypothesis should be applied. In other words, it speculates a lemma which, if it is true, will allow the induction hypothesis to be applied. For example, let's say we are trying to prove $\texttt{rev}\,xs = \texttt{it-rev}\,xs\ \texttt{[ ]}$ by induction on xs, Oyster/Clam will end up blocked trying to prove the following property[2].

[2]This is also where the prover Zeno gets blocked, but as it is lacking a lemma speculation proof critic it is unable to unblock itself and so cannot prove this property automatically.

if	$\texttt{rev}\, xs = \texttt{it-rev}\, xs\ \texttt{[ ]}$	(induction hypothesis)
then	$\texttt{app}\,(\texttt{rev}\, xs)\ \texttt{[}x\texttt{]} = \texttt{it-rev}\, xs\ \texttt{[}x\texttt{]}$	

Based on the shape of the induction hypothesis and the property we are attempting to apply it to, lemma speculation will assume the existence of a lemma with the following shape, where F and G are term meta-variables whose definition the proof critic will attempt to discover.

$$F\,(\texttt{rev}\, xs)\ ys = \texttt{it-rev}\, xs\ (G\ ys)$$

Middle-out reasoning refers to the technique of introducing these term meta-variables into the lemma speculation process, which will stand in for terms the proof critic does not yet know the definition of, but which it expects to exist based on the shape of the lemma we are speculating. Lemma speculation continues by attempting to prove the above property by induction on xs, yielding

if	$\forall ys' \,.\, F\,(\texttt{rev}\, xs)\ ys' = \texttt{it-rev}\, xs\ (G\ ys')$	(induction hypothesis)
then	$F\,(\texttt{app}\,(\texttt{rev}\, xs)\ \texttt{[}x\texttt{]})\ ys = \texttt{it-rev}\, xs\ (x :: G\ ys)$	

Now that the proof is stuck it can attempt to discover a definition of F and G which would allow this proof to be completed and by doing so discover a value of F and G for the originally speculated lemma. In this case, the proof critic appeals to its list of existing lemmas to see if one matches the shape of the goal property above. Let's say we have provided the following property, the associativity of list append: $\texttt{app}\,(\texttt{app}\, xs\ ys)\ zs = \texttt{app}\, xs\ (\texttt{app}\, ys\ zs)$. The lemma speculation critic will instantiate F to be $\texttt{app}$ and apply this to the above goal, allowing it to apply the induction hypothesis.

$$\begin{aligned}
&F\,(\texttt{app}\,(\texttt{rev}\, xs)\ \texttt{[}x\texttt{]})\ ys = \texttt{it-rev}\, xs\ (x :: G\ ys)\\
&\qquad = \{\text{ by instantiating } F = \texttt{app} \ \}\\
&\texttt{app}\,(\texttt{app}\,(\texttt{rev}\, xs)\ \texttt{[}x\texttt{]})\ ys = \texttt{it-rev}\, xs\ (x :: G\ ys)\\
&\qquad = \{\text{ by associativity of } \texttt{app} \ \}\\
&\texttt{app}\,(\texttt{rev}\, xs)\ (\texttt{app}\ \texttt{[}x\texttt{]}\ ys) = \texttt{it-rev}\, xs\ (x :: G\ ys)\\
&\qquad = \{\text{ by definition of } \texttt{app} \ \}\\
&\texttt{app}\,(\texttt{rev}\, xs)\ (x :: ys) = \texttt{it-rev}\, xs\ (x :: G\ ys)\\
&\qquad = \{\text{ by induction hypothesis with } F = \texttt{app} \ \}\\
&\texttt{it-rev}\, xs\ (x :: ys) = \texttt{it-rev}\, xs\ (x :: G\ ys)
\end{aligned}$$

This final step can be trivially proved by choosing $G = \text{fn}\ x.\ x$. Since this proof was completed, it means the original property holds for $F = \texttt{app}$ and $G = \text{fn}\ x.\ x$, so the proof critic can feed these definitions back into our originally speculated lemma:

$$\texttt{app}\,(\texttt{rev}\, xs)\ ys = \texttt{it-rev}\, xs\ ys$$

The above lemma allows Oyster/Clam to complete its original proof of $\texttt{rev}\, xs = \texttt{it-rev}\, xs\ \texttt{[ ]}$, which is a trivial consequence of the above lemma when $ys = \texttt{[ ]}$. The above is not a fully automatic proof however, as it relies on the associativity of `app` having been supplied as a lemma.

This process is fundamentally the same approach as Elea's fold discovery technique from Chapter 8, but while Oyster/Clam appeals to an existing lemma to find the value of an inserted term meta-variable, Elea assumes this term is a fold function, and guesses the definition using its property rewriting system. For example, Elea's proves $\texttt{rev}\, xs = \texttt{it-rev}\, xs\ \texttt{[ ]}$ by rewriting $\texttt{it-rev}\, xs\ \texttt{[ ]} \overset{\sqsubseteq}{\longrightarrow}_{+} \texttt{rev}\, xs$ - see Section 8.1 on page 107. This proof requires finding a definition for the meta-variable F such that:

$$\texttt{it-rev}^{a}\, xs\ \texttt{[}x\texttt{]} \sqsubseteq F\, (\texttt{it-rev}^{a}\, xs\ \texttt{[ ]})$$

Elea guesses that the value of F in the above is $\text{fold}_{\texttt{List}}\langle c_1, c_2\rangle$ for some values of c_1 and c_2. It then uses its theorem prover to manipulate the above inequality until the values of c_1 and c_2 become apparent - Example 8.1 on page 116. The similarities between this and Oyster/Clam's lemma speculation approach is that they both use middle-out reasoning to insert meta-variables to find a lemma which will unblock their proof process. The difference in Elea's approach is that it discovers the definition of F by inventing a potentially entirely new function, though in this case it was the `app` function. Elea can rewrite $\texttt{it-rev}\, xs\ \texttt{[ ]} \overset{\sqsubseteq}{\longrightarrow}_{+} \texttt{rev}\, xs$ without requiring the definition of `rev` or `app`, and can discover both of these definitions using fix-fix fusion and fold-discovery respectively.

The lemma speculation critic is a more general approach in one sense, as it can infer lemmas involving functions which are not fold functions. Elea's fold discovery technique is more general in another sense, as it does not require already provided lemmas in its discovery process.

10.3 Table of results

Now that the test properties have been motivated, and the four theorem provers Elea is compared to have been described, I can present the results of my tool comparison in the table below, structured as follows. The first column gives the identifier of this property, where Z1...Z85 were taken from a test suite originally used by the IsaPlanner tool [32], and P1...P50 are from those originally used to test the Clam proof planner [30]. C1...C8 were chosen by me as properties which could not be proved by induction.

A ✓ indicates that the given tool can prove the given property, and a ✗ indicates that it cannot. The entry n/a means that this property is not applicable to this tool, where n/a for Elea means that this equivalence does not hold as an approximation in either direction, n/a for HOSC means that this property does not hold as an equivalence, and n/a for HipSpec or Zeno means that this property cannot be proved by induction. All approximations are given to HipSpec and Zeno as equivalences, as these tools assume all terms are total.

#	Property	Elea	HOSC	HipSpec	Zeno
Z1	`app (take n xs) (drop n xs)` $\sqsubseteq$ `xs`	✓	n/a	✓	✓
Z2	`add (count n xs) (count n ys)` $\equiv$ `count n (app xs ys)`	✓	✗	✓	✓
Z3	`lq (count n xs) (count n (app xs ys))` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z4	`count n (n :: xs)` $\sqsubseteq$ `Suc (count n xs)`	✓	n/a	✓	✓
Z5	`assert True <- eq n m in Suc (count n xs)` $\equiv$ `count n (m :: xs)`	✓	n/a	✓	✓
Z6	`minus n (add n m)` $\sqsubseteq$ `0`	✓	n/a	✓	✓
Z7	`minus (add n m) n` $\sqsubseteq$ `m`	✓	n/a	✓	✓
Z8	`minus (add k m) (add k n)` $\sqsubseteq$ `minus m n`	✓	n/a	✓	✓
Z9	`minus (minus n m)` $\equiv$ `minus n (add m k)`	✓	✗	✓	✓
Z10	`minus n n` $\sqsubseteq$ `0`	✓	n/a	✓	✓
Z11	`drop 0 xs` $\equiv$ `xs`	✓	✓	✓	✓
Z12	`drop n (map f xs)` $\equiv$ `map f (drop n xs)`	✓	✓	✓	✓
Z13	`drop (Suc n) (Cons x xs)` $\equiv$ `drop n xs`	✓	✓	✓	✓
Z14	`filter p (app xs ys)` $\equiv$ `app (filter p xs) (filter p ys)`	✓	✓	✓	✓
Z15	`len (insert n xs)` $\sqsubseteq$ `Suc (len xs)`	✓	n/a	✓	✓
Z16	`assert True <- null xs in last (x :: xs)` $\equiv$ `x`	✓	n/a	✓	✓
Z17	`assert True <- lq n 0 in n` $\equiv$ `0`	✓	n/a	✓	✓

#	Property	Elea	HOSC	HipSpec	Zeno
Z18	`lt n (Suc (add n m))` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z19	`len (drop n xs)` $\equiv$ `minus (len xs) n`	✓	✗	✓	✓
Z20	`len (isort xs)` $\sqsubseteq$ `len xs`	✓	n/a	✓	✓
Z21	`lq n (add n m)` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z22	`max (max n m) k` $\equiv$ `max n (max m k)`	✓	✓	✓	✓
Z23	`max n m` $\equiv$ `max m n`	n/a	n/a	✓	✓

Z24	`assert True <- eq (max n m) n in`
	`lq m n` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z25	`assert True <- eq (max n m) m in`
	`lq n m` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z26	`assert True <- elem n xs in`
	`elem n (app xs ys)` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z27	`assert True <- elem n ys in`
	`elem n (app xs ys)` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z28	`elem n (app xs [n])` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z29	`elem n (eq_insert n xs)` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z30	`elem n (lt_insert n xs)` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z31	`min (min n m) k` $\equiv$ `min n (min m k)`	✓	✓	✓	✓
Z32	`min n m` $\equiv$ `min m n`	n/a	n/a	✓	✓

Z33	`assert True <- eq (min n m) n in`
	`lq n m` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z34	`assert eq (min n m) m in`
	`lq m n` $\sqsubseteq$ `True`
		✓	n/a	✓	✓

Z35	`dropWhile (fn x -> False) xs` $\equiv$ `xs`	✓	✗	✓	✓
Z36	`takeWhile (fn x -> True) xs` $\equiv$ `xs`	✓	✗	✓	✓
Z37	`not (elem n (delete n xs))` $\sqsubseteq$ `True`	✓	n/a	✓	✓
Z38	`count n (app xs [n])` $\sqsubseteq$ `Suc (count n xs)`	✓	n/a	✓	✓

#	Property	Elea	HOSC	HipSpec	Zeno
Z39	`add (count n [x]) (count n xs)` `≡ count n (x :: xs)`	✓	✗	✓	✓
Z40	`take 0 xs ≡ []`	✓	✓	✓	✓
Z41	`take n (map f xs)` `≡ map f (take n xs)`	✓	✓	✓	✓
Z42	`take (Suc n) (x :: xs)` `≡ x :: (take n xs)`	✓	✓	✓	✓
Z43	`app (takeWhile p xs)` `(dropWhile p xs) ⊑ xs`	✓	n/a	✓	✓
Z44	`zip (x :: xs) ys` `⊑ zipConcat x xs ys`	✓	n/a	✓	✓
Z45	`zip (x :: xs) (y :: ys)` `≡ (x, y) :: (zip xs ys)`	✓	✓	✓	✓
Z46	`zip [] xs ≡ []`	✓	✗	✓	✓
Z47	`height (mirror t) ≡ height t`	n/a	n/a	✓	✓
Z48	`assert False <- null xs in` `app (butlast xs) ([last xs]) ≡ xs`	✓	n/a	✓	✓
Z49	`butlast (app xs ys)` `≡ butlastConcat xs ys`	n/a	n/a	✓	✓
Z50	`butlast xs` `≡ take (minus (len xs) (Suc 0)) xs`	✓	✗	✗	✓
Z51	`butlast (app xs [x]) ⊑ xs`	✓	n/a	✓	✓
Z52	`count n xs ≡ count n (rev xs)`	n/a	n/a	✓	✓
Z53	`count n (isort xs) ≡ count n xs`	n/a	n/a	✓	✓
Z54	`minus (add m n) n ⊑ m`	✗	n/a	✓	✓
Z55	`app (drop n xs)` `(drop (minus n (len xs)) ys)` `≡ drop n (app xs ys)`	✓	✗	✓	✓

#	Property	Elea	HOSC	HipSpec	Zeno
Z56	`drop n (drop m xs)` ≡ `drop (add m n) xs`	✓	✗	✓	✓
Z57	`drop n (take m xs)` ≡ `take (minus m n) (drop n xs)`	n/a	n/a	✓	✓
Z58	`drop n (zip xs ys)` ≡ `zip (drop n xs) (drop n ys)`	✓	✗	✓	✓
Z59	`assert True <- null ys in` `last (app xs ys)` ≡ `last xs`	✓	n/a	✓	✓
Z60	`assert False <- null ys in` `last (app xs ys)` ≡ `last ys`	✓	n/a	✓	✓
Z61	`last (app xs ys)` ≡ `lastOfTwo xs ys`	n/a	✗	✓	✓
Z62	`assert False <- null xs in` `last (x :: xs)` ≡ `last xs`	✓	n/a	✓	✓
Z63	`assert True <- lt n (len xs) in` `last (drop n xs)` ⊑ `last xs`	✓	n/a	✓	✓
Z64	`last (app xs [x])` ⊑ `x`	✓	n/a	✓	✓
Z65	`lt n (Suc (add m n))` ⊑ `True`	✓	n/a	✓	✓
Z66	`lq (len (filter p xs)) (len xs)` ⊑ `True`	✓	n/a	✗	✓
Z67	`len (butlast xs)` ⊑ `minus (len xs) (Suc 0)`	✓	n/a	✓	✓
Z68	`lq (len (delete n xs)) (len xs)` ⊑ `True`	✓	n/a	✗	✓
Z69	`lq n (add m n)` ⊑ `True`	✓	n/a	✓	✓
Z70	`assert True <- lq m n in` `lq m (Suc n)` ⊑ `True`	✓	n/a	✓	✓
Z71	`assert False <- eq n m in` `elem n (insert m xs)` ⊑ `elem n xs`	✓	n/a	✓	✓
Z72	`rev (drop n xs)` ≡ `take (minus (len xs) n) (rev xs)`	✗	n/a	✓	✗

#	Property	Elea	HOSC	HipSpec	Zeno
Z73	`rev (filter p xs)` ≡ `filter p (rev xs)`	n/a	n/a	✓	✓
Z74	`rev (take n xs)` ≡ `drop (minus (len xs) n) (rev xs`	✗	n/a	✓	✗
Z75	`add (count n xs) (count n [m])` ≡ `count n (m :: xs)`	n/a	n/a	✓	✓

Z76	`assert False <- eq n m in`
	 `count n (app xs [m])` ⊑ `count n xs`
		✓	n/a	✓	✓

Z77	`assert True <- sorted xs in`
	 `sorted (insert n xs)` ⊑ `True`
		✓	n/a	✓	✓

Z78	`sorted (isort xs)` ⊑ `True`	✓	n/a	✗	✓
Z79	`minus (minus m n) k` ≡ `minus (minus (Suc m) n) (Suc k)`	n/a	n/a	✓	✓

Z80	`app (take n xs)`
	 `(take (minus n (len xs)) ys)`
	 ≡ `take n (app xs ys)`
		✓	✗	✓	✓

Z81	`drop m (take (add n m) xs)` ≡ `take n (drop m xs)`	✗	✗	✓	✓
Z82	`take n (zip xs ys)` ≡ `zip (take n xs) (take n ys)`	✓	✓	✓	✓

Z83	`app (zip xs (take (len xs) zs))`
	 `(zip ys (drop (len xs) zs))`
	 ≡ `zip (app xs ys) zs`
		✓	✗	✓	✓

Z84	`app (zip (take (len ys) xs) ys)`
	 `(zip (drop (len ys) xs) zs)`
	 ≡ `zip xs (app ys zs)`
		✓	✗	✓	✓

Z85	`assert True <- eq (len xs) (len ys) in  rev (zip`
	 `xs ys)`
	 ≡ `zip (rev xs) (rev ys)`
		✗	n/a	✗	✗

#	Property	Elea	HOSC	HipSpec	Zeno	Clam
P1	`add n n` $\sqsubseteq$ `double n`	✓	n/a	✓	✗	✓
P2	`len (app xs ys)` $\equiv$ `len (app ys xs)`	n/a	n/a	✓	✓	✓
P3	`len (app xs ys)` $\equiv$ `add (len ys) (len xs)`	n/a	n/a	✓	✓	✓
P4	`len (app xs xs)` $\sqsubseteq$ `double (len xs)`	✓	n/a	✓	✗	✓
P5	`len (rev xs)` $\sqsubseteq$ `len xs`	✓	n/a	✓	✓	✓
P6	`len (rev (app xs ys))` $\sqsubseteq$ `add (len xs) (len ys)`	✓	n/a	✓	✓	✓
P7	`len (it-rev xs ys)` $\sqsubseteq$ `add (len xs) (len ys)`	✓	n/a	✓	✓	✓
P8	`drop n (drop m xs)` $\equiv$ `drop m (drop n xs)`	n/a	n/a	✓	✗	✓
P9	`drop n (drop m (drop k xs))` $\equiv$ `drop k (drop m (drop m xs))`	n/a	n/a	✓	✗	✓
P10	`rev (rev xs)` $\sqsubseteq$ `xs`	✓	n/a	✓	✓	✓
P11	`rev (app (rev xs) (rev ys))` $\sqsubseteq$ `app ys xs`	✗	n/a	✓	✗	✓
P12	`it-rev xs ys` $\equiv$ `app (rev xs) ys`	✓	n/a	✓	✓	✓
P13	`half (add n n)` $\sqsubseteq$ `n`	✓	n/a	✓	✗	✓
P14	`sorted (isort xs)` $\sqsubseteq$ `True`	✓	n/a	✗	✓	✓
P15	`add n (Suc n)` $\sqsubseteq$ `Suc (add n n)`	✓	n/a	✓	✗	✓
P16	`even (add n n)` $\sqsubseteq$ `True`	✓	n/a	✓	✗	✓
P17	`rev (rev (app xs ys))` $\sqsubseteq$ `app (rev (rev xs)) (rev (rev ys))`	✗	n/a	✓	✗	✓
P18	`rev (app (rev xs) ys)` $\sqsubseteq$ `app (rev ys) xs`	✗	n/a	✓	✗	✓
P19	`rev (rev (app xs ys))` $\sqsubseteq$ `app (rev (rev xs)) ys`	✓	n/a	✓	✗	✓
P20	`even (len (app xs xs))` $\sqsubseteq$ `True`	✓	n/a	✓	✗	✓

#	Property	Elea	HOSC	HipSpec	Zeno	Clam
P21	`rotate (len xs) (app xs ys)` $\sqsubseteq$ `app ys xs`	✗	n/a	✓	✗	✓
P22	`even (len (app xs ys))` $\equiv$ `even (len (app ys xs))`	✗	✗	✓	✓	✓
P23	`half (length (app xs ys))` $\equiv$ `half (length (app ys xs))`	n/a	n/a	✓	✓	✓
P24	`even (add n m)` $\equiv$ `even (add m n)`	✗	✗	✓	✓	✓
P25	`even (len (app xs ys))` $\equiv$ `even (add (len ys) (len xs))`	✗	n/a	✓	✓	✓
P26	`half (add n m)` $\equiv$ `half (add m n)`	n/a	n/a	✓	✓	✓
P27	`rev xs` $\equiv$ `it-rev xs []`	✓	✗	✓	✗	✗
P28	`revflat xs` $\equiv$ `it-revflat xs Nil`	✓	✗	✓	✗	✗
P29	`rev (qrev xs Nil)` $\sqsubseteq$ `xs`	✓	n/a	✓	✗	✗
P30	`rev (app (rev xs) Nil)` $\sqsubseteq$ `xs`	✓	n/a	✓	✓	✗
P31	`it-rev (it-rev xs Nil) Nil` $\sqsubseteq$ `xs`	✓	n/a	✓	✗	✗
P32	`rotate (len xs) xs` $\sqsubseteq$ `xs`	✗	n/a	✓	✗	✗
P33	`fac n` $\equiv$ `it-fac n (Suc 0)`	✗	✗	✓	✗	✗
P34	`mul n m` $\equiv$ `it-mul n m 0`	✓	✗	✓	✗	✗
P35	`exp x y` $\equiv$ `it-exp x y (Suc 0)`	✗	✗	✓	✗	✗
P36	`assert True <- elem n xs in` `elem n (app xs ys)` $\sqsubseteq$ `True`	✓	n/a	✓	✓	✓
P37	`assert True <- elem n ys in` `elem n (app xs ys)` $\sqsubseteq$ `True`	✓	n/a	✓	✓	✓
P38	`assert True <- elem n xs in` `assert True <- elem n ys in` `elem n (app xs ys)` $\sqsubseteq$ `True`	✓	n/a	✓	✓	✓
P39	`assert True <- elem n (drop m xs) in` `elem n xs` $\sqsubseteq$ `True`	✓	n/a	✓	✓	✓

#	Property	Elea	HOSC	HipSpec	Zeno	Clam

P40	`assert True <- subset xs ys in`
	 `union xs ys` $\sqsubseteq$ `ys`
		✓	n/a	✓	✗	✓

P41	`assert True <- subset xs ys in`
	 `intersect xs ys` $\sqsubseteq$ `xs`
		✓	n/a	✓	✗	✓

P42	`assert True <- elem n xs in`
	 `elem n (union xs ys)` $\sqsubseteq$ `True`
		✓	n/a	✗	✗	✓

P43	`assert True <- elem n ys in`
	 `elem n (union xs ys)` $\sqsubseteq$ `True`
		✓	n/a	✓	✗	✓

P44	`assert True <- elem n xs in`
	 `assert True <- elem n ys in`
	 `elem n (intersect xs ys)` $\sqsubseteq$ `True`
		✓	n/a	✗	✗	✓

P45	`elem n (insert n xs)` $\sqsubseteq$ `True`	✓	n/a	✓	✓	✓

P46	`assert True <- eq n m in`
	 `elem n (insert m xs)` $\sqsubseteq$ `True`
		✓	n/a	✓	✗	✓

P47	`assert False <- eq n m) in`
	 `elem n (insert m xs)` $\sqsubseteq$ `elem n xs`
		✓	n/a	✓	✗	✓

P48	`len (isort xs)` $\sqsubseteq$ `len xs`	✓	n/a	✓	✓	✓

P49	`assert True <- elem n (isort xs) in`
	 `elem n xs` $\sqsubseteq$ `True`
		✓	n/a	✓	✗	✓

P50	`count n (isort xs)` $\equiv$ `count n xs`	n/a	n/a	✓	✓	✓

#	Property	Elea	HOSC	HipSpec	Zeno
C1	`map f (repeat n)` $\equiv$ `repeat (f n)`	✓	✓	n/a	n/a
C2	`tail (iterate f x)` $\equiv$ `iterate f (f x)`	✓	✓	n/a	n/a
C3	`map f (iterate f x)` $\equiv$ `iterate f (f x)`	✓	✓	n/a	n/a
C4	`filter p (repeat x)` $\sqsubseteq$ `repeat x`	✓	n/a	n/a	n/a
C5	`drop n (repeat x)` $\equiv$ `repeat x`	✓	✗	n/a	n/a
C6	`butlast (iterate f x)` $\equiv$ `iterate f x`	✓	✗	n/a	n/a
C7	`last (iterate f x)` $\equiv$ $\bot$	✓	✗	n/a	n/a
C8	`sorted (repeat x)` $\equiv$ $\bot$	✓	✗	n/a	n/a

10.4 Summary and analysis

Below is a table summarising how many properties were applicable to each tool, and how many of these properties each could prove. After this I give an analysis of these results.

Tool	Applicable	Proved	Unproved	Success rate
Elea	125	110	15	88%
HOSC	41	14	27	34%
HipSpec	135	127	8	94%
Zeno	135	103	32	76%
Clam	50	41	9	82%

HipSpec is the most successful tool within its domain, able to prove all but 8 of the 135 properties applicable to it. All but one (Z50) of the 7 properties Elea could prove over HipSpec require a lemma with an implication, something HipSpec is unable, currently, to conjecture.

HOSC is the least successful, largely due to it only checking for α-equality after supercompilation. Properties such as Z35 and Z36 would be easy for it to prove with a stronger equality check, such as applying the least fixed-point principle bidirectionally.

Zeno, being a top-down induction prover, suffers heavily from the generalisation problem, as the introduction to this thesis discusses, and as is evidenced by this test set. Many properties from the P1...P50 test set require a non-trivial generalisation of a sub-goal, which Zeno is unable to guess. Elea was able to prove many more of these properties than Zeno, though some generalisations were beyond the scope of Elea's rewriting algorithm, such as those required to show P21, P33 and P35. HipSpec's bottom-up approach suffers the least from the generalisation problem, and was able to prove these properties.

Clam performed very well on it's test set P1..P50, missing a proof of only 9 properties. These 9 properties are provable by the Oyster/Clam system, but they each require a lemma to be provided to inspire Clam's lemma speculation step (Section 10.2.4 on page 138), and hence are not provable fully automatically.

Of the properties Elea is unable to prove, Z3, Z54, Z72, Z74, Z85, P21, P33, and P35 were due to its rewriting system being unable to find a fixed-point promoted form for the term on the left-hand side of the approximation, and hence being unable to apply the least fixed-point principle. That is to say, Elea wasn't smart enough to prove the property. Properties P21, P33, and P35, in particular, are a failing of the fold discovery method to find the appropriate fold function which would allow Elea to produce a term in fixed-point promoted form. Its failure to prove P11, P17, and P18, however, is due to the rewriting system producing terms which were too defined on the left-hand side of the approximation, and invalidating the property. For example, in P17, `rev (rev (app xs ys))` $\sqsubseteq$ `app (rev (rev xs)) (rev (rev ys))`, Elea rewrites the left-hand side to `app xs ys`, rendering the property false.

The properties P22, P24, and P25, while true for non-total terms, rely on the the same shape of proof as if we were proving commutativity of addition, something which is not true for non-total terms. More specifically, the top-down proof of these properties all rely on an internal proof of $\texttt{add}\ x\ (\texttt{Suc}\ (\texttt{Suc}\ y)) \sqsubseteq \texttt{Suc}\ (\texttt{Suc}\ (\texttt{add}\ x\ y))$, a step Elea cannot make, as this property does not hold for non-total terms.

10.5 Using seq to extend Elea

The end of previous section explained that Elea was unable to prove certain properties because, while these properties might hold for non-total terms, the internal proof steps a top-down prover will take to prove them do not hold for non-total terms . For example, to prove `rev (rev (app xs ys))` $\sqsubseteq$ `app (rev (rev xs)) (rev (rev ys))`, Elea proves `rev (rev (app xs ys))` $\sqsubseteq$ `app xs ys`, reducing its goal to `app xs ys` $\sqsubseteq$ `app (rev (rev xs)) (rev (rev ys))`, which no longer holds! Similarly, in trying to prove `even (add n m)` $\equiv$ `even (add m n)`, the internal proof steps Elea would need to take, resemble a proof of `add n m` $\equiv$ `add m n`, which does not hold unless we assume totality.

I have an idea as to how properties such as this could be proved within Elea, by extending the fission steps from Chapter 6 so that they preserve equivalence. This idea uses the seq syntax defined on page 24. Taking subterm fission as an example, currently subterm fission performs rewrites such as

$$\text{fix}\begin{pmatrix}\text{fn } f, n.\ \text{case } n \text{ of} \\ \quad \texttt{0} \to \texttt{True} \\ \quad \texttt{Suc}\ n' \to f\ n'\end{pmatrix} \xrightarrow{\sqsubseteq}_{+} \text{fn } n.\ \texttt{True}$$

The above is not an equivalence preserving rewrite, since if the argument n contains undefinedness the left-hand term will be undefined, whereas the right-hand term will still be `True`. This rewrite has lost that the n argument has been fully recursed over. My idea is to restore this information, and turn rewrites like the above into equivalence preserving rewrites.

This extension relies on a new bit of syntax, which I call foldseq This syntax can be defined for any data-type in νPCF, and for a given data-type $\boldsymbol{T}$ and any type τ, $\text{foldseq}_{\boldsymbol{T}} : \boldsymbol{T} \to \tau \to \tau$. The purpose of this function is to fully evaluate the structure of the first argument before returning the second. So, for any x, $\text{foldseq}_{\texttt{Nat}}\ (\texttt{Suc}\ \bot)\ x \equiv \bot$, but $\text{foldseq}_{\texttt{Nat}}(\texttt{Suc}\ \texttt{0})\ x \equiv x$.

Below are the definitions of foldseq for the `Nat` and `list` data-types.

$$\begin{aligned}
\mathrm{foldseq}_{\mathtt{Nat}}\, n\, x &\stackrel{\mathrm{def}}{=} \mathrm{seq}\,(\mathrm{fold}_{\mathtt{Nat}}\langle \mathtt{unit}, \mathrm{fn}\ n'.\, n'\rangle\ n)\ \mathrm{in}\ x \\
\mathrm{foldseq}_{\mathtt{list}}\, ys\, x &\stackrel{\mathrm{def}}{=} \mathrm{seq}\,(\mathrm{fold}_{\mathtt{list}}\langle \mathtt{unit}, \mathrm{fn}\ y, ys'.\, ys'\rangle\ ys)\ \mathrm{in}\ x
\end{aligned}$$

This syntax is very similar to the `deepseq` function used within Haskell [60], the difference being that foldseq only evaluates the outer data-structure, whereas `deepseq` also fully evaluates all terms within the data-structure.

Using foldseq, we may be able to extend subterm fission to perform the following rewrite:

$$\mathrm{fix}\begin{pmatrix}\mathrm{fn}\ f, n.\ \mathrm{case}\ n\ \mathrm{of} \\ \quad \mathtt{0} \to \mathtt{True} \\ \quad \mathtt{Suc}\, n' \to f\ n'\end{pmatrix} \xrightarrow{\sqsubseteq}_{+} \mathrm{fn}\ n.\ \mathrm{foldseq}_{\mathtt{Nat}}\, n\, \mathtt{True}$$

This rewrite is now equivalence preserving, as it has kept that n is fully evaluated before `True` is returned. Another non-equivalence preserving rewrite fission currently performs is the one used within the rewrite of $\mathtt{rev}\ (\mathtt{rev}\, xs) \xrightarrow{\sqsubseteq}_{+} xs$:

$$\mathrm{fix}\begin{pmatrix}\mathrm{fn}\ f, xs.\ \mathrm{case}\ xs\ \mathrm{of} \\ \quad \mathtt{[\,]} \to \mathtt{[}y\mathtt{]} \\ \quad x :: xs' \to \mathtt{snoc}_x\ (f\ xs')\end{pmatrix} \xrightarrow{\sqsubseteq}_{+} \mathrm{fn}\ xs.\ y :: \mathtt{rev}\, xs$$

This rewrite has lost that xs is fully evaluated before the y argument is given as the head of the list. Again, we could use foldseq to attempt to recover equivalence, turning this rewrite into

$$\mathrm{fix}\begin{pmatrix}\mathrm{fn}\ f, xs.\ \mathrm{case}\ xs\ \mathrm{of} \\ \quad \mathtt{[\,]} \to \mathtt{[}y\mathtt{]} \\ \quad x :: xs' \to \mathtt{snoc}_x\ (f\ xs')\end{pmatrix} \xrightarrow{\sqsubseteq}_{+} \mathrm{fn}\ xs.\ \mathrm{foldseq}_{\mathtt{list}}\, xs\ (y :: \mathtt{rev}\, xs)$$

Using this fission step, we could amend the $\mathtt{rev}\ (\mathtt{rev}\ xs) \xrightarrow{\sqsubseteq}_{+} xs$ rewrite into $\mathtt{rev}\ (\mathtt{rev}\, xs) \xrightarrow{\sqsubseteq}_{+} \mathrm{foldseq}\, xs\, xs$, which now preserves equivalence.

Chapter 11

Related work

This chapter overviews the existing literature related to this thesis, excluding the theorem provers Zeno, HipSpec, HOSC, and Oyster/Clam, as these have already been detailed in Section 10.2. In Section 2.4.2 on page 42 I detailed the general shape of unfold-fold transformations, a class which includes Elea's fusion rewrite steps. Specific implementations of unfold-fold are explained in this chapter, including supercompilation (Section 11.1), deforestation (Section 11.2), and generalised partial computation (Section 11.3). This chapter then goes on to explain two other methods used to prove unfold-fold style rewrites are sound, improvement theory (Section 11.4) and bisimulation (Section 11.4).

11.1 Supercompilation

Supercompilation is arguably most used of the unfold-fold rewriting techniques. Originally developed by Turchin [68, 69] for the Refal language, it was later reformulated by Sørensen et al. [66] for a more traditional, first-order, functional language. The latter reformulation was into a positive supercompiler, terminology which refers to fact that the algorithm only propagates positive information when descending into a term. Positive information refers to equivalences, or potential substitutions, between terms, whereas negative information refers to which terms are not equivalent.

Elea, in one sense, also propagates only positive information, as the facts stored in Φ by the traverse branch rule (page 59), are positive. However, since a fact could represent a pattern match of a predicate to `False`, such as `False` $\sqsubseteq$ `eq` $x\ y$, it can also, in another sense, propagate negative information.

Modern supercompilers ensure termination using the same method as Elea, by an online test for homeomorphic embedding, or a similar well-quasi-order, on an environment set of previously encountered terms.

One of the distinguishing features of supercompilation is its generalisation heuristic. While no two supercompilers use exactly the same method, the methods they do use are all based upon the most-specific generalisation technique [65]. Unlike Elea, which immediately generalises terms to isolate individual fixed-point promoted form anti-patterns, this method defers generalisation until a folding step is blocked. Upon reaching such a blockage, the algorithm takes the folding step we are trying to apply, and uses it to generalise the blocked term. The hope is that further driving will become applicable after this generalisation, such that the folding step can then be applied. This process is similar to the critical paths based generalisation technique used by the Zeno theorem prover [64], and the proof critics used by IsaPlanner [21] and Oyster/Clam [14].

I will now give two examples of specific supercompilers. Bolingbroke's supercompiler [6] is designed to optimise call-by-need languages, particularly Haskell. One unique feature of this supercompiler is that it annotates every node of the expression tree of a term with a unique identifier, called a tag. These tags are used for multiple purposes [7], including an online termination check based on comparing the sets of tags in each term. This termination check is much faster than the homeomorphic embedding, and so is more useful for building a practical program optimising supercompiler, but blocks more potential rewrites, and so would be less useful for theorem proving. I investigated this tag-bag approach when developing Elea, and found that it would block many rewrites required to prove the properties from Chapter 10.

The HOSC supercompiler [40, 39, 43, 41], already discussed in Section 10.2.1, uses supercompilation to prove term equivalence, by checking for α-equality of terms after supercompilation. Version 1.5 of HOSC [42] features an extension of the most-specific generalisation heuristic to allow for more provable properties which feature higher-order functions. The pre-generalisation technique used by Elea's fusion rewrites is also able to prove the properties given as examples in this paper.

Recently, supercompilation has been extended into distillation by Hamilton [28]. This method generalises the unification step within the folding rewrite to also unify functions whose recursive definitions match, rather than only unifying functions with matching names. Elea gets this feature for free, as all terms in νPCF are anonymous. Furthermore, the example given in Hamilton's paper, the rewriting of `app` (`it-rev` xs `[ ]`) ys into `it-rev` $xs\,ys$, will also be performed by Elea.

11.2 Deforestation

Deforestation is another unfold-fold technique, designed as an optimisation technique which could be included in real-world language compilers. For example, a variant of deforestation is included in the GHC compiler for Haskell [67]. Originally due to Wadler [72], it ensures both soundness and termination by a syntactic restriction on which programs it is applicable to. This restriction is referred to as treeless form.

Treeless form is a fast syntactic check which is very useful for practical program optimisation, but too restrictive for the rewrites Elea requires in order to prove the properties in Chapter 10, despite later research which broadens its applicability [73, 27]. It initially was not applicable to higher-order functions, but has since been extended to remove this restriction [51, 25].

11.3 Generalised partial computation

One unfold-fold technique of particular relevance to Elea is generalised partial computation, or GPC [23, 22]. GPC uses the unfold-fold principle to remove unreachable branches of recursive functions, and can collapse a recursive function into a non-recursive term, if every reachable branch is equivalent to this term. GPC detects these unreachable branches by calling an external theorem prover, using a knowledge database about built-in predicates such as $\leq$.

This method inspired Elea's fact fusion rewrite step in Section 5.9, which also removes unreachable branches of recursive functions, and is often followed by subterm fission (page 90) to collapse recursive functions into non-recursive terms. However, the goal of Elea is to automatically prove approximations from only function definitions, and as such its input language has no built in predicates, so the GPC approach was not directly applicable. Instead, fact fusion uses Elea as a theorem prover within itself, in order to reason about user-defined predicates, instead of only those which are built-in.

11.4 Improvement theory

Sands' improvement theory [61, 62, 63] is a method for proving the unfold-fold principle preserves $\sqsubseteq$, used by both HOSC and Bolingbroke's supercompiler.

Improvement theory can be stated as, if our driving rewrite preserves $\sqsubseteq$, and if the complexity of the result of driving is less than the original term, then unfold-fold preserves $\sqsubseteq$. In this context, the complexity of A is said to be less than B, if for any closing program context $\mathcal{C}$, the number of fixed-points which must be unrolled to evaluate $\mathcal{C}[A]$ to a value must be less than the number required to fully evaluate $\mathcal{C}[B]$. Referring back to the unfold-fold principle given in Section 2.4.2:

$$\frac{h := \text{fn } x_1, ..., x_n.\, B \;\;\vdash\;\; \text{fn } x_1, ..., x_n.\, B \longrightarrow H}{\vdash\;\; B[A_1/x_1]...[A_n/x_n] \longrightarrow \text{fix}\,(\text{fn } h.\, H)\, A_1...A_n}$$

In the above, improvement theory requires fn $x_1, ..., x_n.\, B \sqsubseteq H$, and that H must less computationally complex than fn $x_1, ..., x_n.\, B$, but allowing us to use $h =$ fn $x_1, ..., x_n.\, B$ as an assumption within this rewrite. If our driving rewrite ensures this, we can conclude $B[A_1/x_1]...[A_n/x_n] \sqsubseteq \text{fix}\,(\text{fn } h.\, H)\, A_1...A_n$, and also that the complexity of $\text{fix}\,(\text{fn } h.\, H)\, A_1...A_n$ is not greater than that of $B[A_1/x_1]...[A_n/x_n]$. If our unfold-fold driving step always starts by unfolding a recursive function, as in supercompilation, then we only need to check that computational complexity is preserved by driving.

Preserving, or decreasing, complexity, is always desired in program optimising rewrite techniques, like Bolingbroke's supercompiler, but in theorem proving we sometimes need to rewrite to a term which is more computationally expensive than the original. A good example of this comes from the two definitions of the list reversal function, `rev` and `it-rev`, both given in Appendix A. Given these definitions, the following property is very easy to prove for automated induction provers:

$$\texttt{rev}\,(\texttt{rev}\, xs) \sqsubseteq xs$$

In contrast, this property is very difficult to automatically prove:

$$\texttt{it-rev}\,(\texttt{it-rev}\, xs\ \texttt{[ ]})\ \texttt{[ ]} \sqsubseteq xs$$

The only automated tool, before mine, which can prove the above purely from its definition is HipSpec [18], which it does by conjecturing and proving the lemma $\forall xs \,.\, \texttt{rev}\; xs = \texttt{it-rev}\; xs\; \texttt{[ ]}$, and then using it to rewrite $\texttt{it-rev}\;(\texttt{it-rev}\; xs\; \texttt{[ ]})\; \texttt{[ ]}$ to $\texttt{rev}\;(\texttt{rev}\; xs)$, yielding the easier property[1]. Elea also proves the above by rewriting `it-rev` to `rev`, something which would be impossible if I used improvement theory to show my method to be sound, as $\texttt{rev}\; xs$ is computationally more expensive than $\texttt{it-rev}\; xs\; \texttt{[ ]}$ for every value of xs except `[ ]`.

11.5 Bisimulation

Another approach to showing unfold-fold preserves equivalence, is to view terms as a transition system and show that unfold-fold preserves equivalence of this system up to bisimulation, an approach due to Hamilton [26]. This method as it stands would be unsuitable for Elea, as there are multiple rewrite steps which preserve denotational approximation, but alter semantics if terms are viewed as a transition system. My fission rules from Chapter 6 would be particularly problematic. Unlike improvement theory, there is no fundamental reason why a bisimulation based proof of soundness would not have been applicable, but developing one would have required a lot of additional research, and I found my approach of using denotational semantics and truncated fixed-points to be much simpler.

11.6 The constructive ω rule

One of the virtues I have extolled of the fixed-point promotion approach to theorem proving is that it simplifies the problem of generalisation. Another technique for proving equivalence properties of functional terms which also simplifies this issue is the work of Siani Baker on automated proof using the constructive ω rule in place of induction [4, 5]. Given a property $P \subseteq \mathbb{N}$, the constructive ω rule for this property is:

$$\frac{P(0), P(1), P(2)...}{\forall (n \in \mathbb{N}) \,.\, P(n)}$$

The above states that if we have meta-level function which is able to generate an instance of a proof of $P(n)$ for every natural number n, then we have a proof of $\forall (n \in \mathbb{N}) \,.\, P(n)$. In essence it has lifted the $\forall$ quantifier from the property term level into being a proof generating function argument at the meta-level. As discussed in "What is a proof" [11], these constructive proofs at the meta-level are not isomorphic to proofs at the term level, and are often easier. For example, let's define $+$ with the following two equations:

$$\begin{array}{lcll} 0 + y & = & y & (1) \\ (\texttt{Suc}\; x) + y & = & \texttt{Suc}\;(x + y) & (2) \end{array}$$

[1]HipSpec proves this property as an equivalence, and assumes that all terms are total.

Using this definition, here is an example proof which utilises the constructive ω rule.

$$\begin{array}{l}
\forall(x \in \mathbb{N}) \,.\, (x + x) + x = x + (x + x) \\
\qquad \Leftarrow \{ \text{ by the constructive } \omega \text{ rule } \} \\
(\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0) + \mathtt{Suc}^n\ 0 = \mathtt{Suc}^n\ 0 + (\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0) \\
\qquad \Leftrightarrow \{ \text{ by applying (2) } n \text{ times on both sides } \} \\
\mathtt{Suc}^n\ (0 + \mathtt{Suc}^n\ 0) + \mathtt{Suc}^n\ 0 = \mathtt{Suc}^n\ (0 + (\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0)) \\
\qquad \Leftrightarrow \{ \text{ by applying (1) on both sides } \} \\
\mathtt{Suc}^{2n}\ 0 + \mathtt{Suc}^n\ 0 = \mathtt{Suc}^n\ (\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0) \\
\\
\qquad \Leftrightarrow \{ \text{ by applying (1) } n \text{ times on the left } \} \\
\mathtt{Suc}^n\ (\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0) = \mathtt{Suc}^n\ (\mathtt{Suc}^n\ 0 + \mathtt{Suc}^n\ 0) \\
\qquad \Leftrightarrow \{ \text{ by reflexivity of equality } \} \\
\top
\end{array}$$

The first step in this proof represents a lifting of x from the term level into natural number n at the meta-level, then the rest of the proof proceeds using meta-level reasoning. Notice that this proof does not require any generalisation of xs to be completed, unlike the equivalent proof in an automated induction prover, such as Zeno or Oyster/Clam.

Research has been conducted into using these meta proofs to infer the lemmas required for term-level induction proofs [3] and there is potential for future work in investigating unfold-fold rewriting techniques which use the constructive ω rule to guide the rewriting process. Perhaps this would have no gain for theorem proving, as the constructive ω rule already simplifies the problem of generalisation and so the advantages of fixed-point promotion may be nullified, but there could be advances to be made in using constructive ω guided unfold-fold rewrite rules to reduce program complexity.

Chapter 12

Conclusion

This thesis has described Elea, a tool for automatically proving properties of denotational, and hence observational, approximation between terms in a functional language with non-strict data-types. The introduction to this thesis argued for the necessity of proving approximation, instead of equivalence, in the presence of non-strictness. This has been reinforced in our evaluation of Elea in Chapter 10, as 68% of the properties tested only hold as an approximation in one direction, and would not have been applicable to Elea if it were only able to prove equivalence.

The technique Elea uses to prove approximation properties is a novel method I have called fixed-point promotion, outlined in Chapter 3. The evaluation of Elea has shown this technique to be very effective, as the properties Elea was unable to prove were due to weaknesses of its term rewriting system, not of the fixed-point promotion technique as a whole. This is to say, there is no fundamental reason why fixed-point promotion cannot prove these properties, we would need only to extend Elea's term rewriting system sufficiently. Elea is also able to use fixed-point promotion to prove properties of codata, something which is impossible for induction, the proof method used by many existing tools.

As discussed in the introduction, the problem of generalisation for fixed-point promotion is far easier than the problem of generalisation for automated top-down cyclic provers. This has been evidenced by Elea's ability to prove many properties which require non-trivial generalisation steps in Chapter 10. This benefit is not without cost though, and the problem of lemma discovery for rewriting based proof techniques, such as fixed-point promotion, is far more difficult than for cyclic provers.

Therefore, much of the research in this thesis has been the development of term rewriting steps which can simulate the lemma discovery techniques used by automated top-down cyclic provers. For example, the fusion steps from Chapter 5 can rewrite $\texttt{add}\,x\,x$ to $\texttt{double}\,x$, a rewrite which relies on the fission rewrites from Chapter 6 to transform $\texttt{add}\,x\,(\texttt{Suc}\,x)$ into $\texttt{Suc}\,(\texttt{add}\,x\,x)$. This mimics the proof a top down cyclic tool would construct of $\texttt{add}\,x\,x \sqsubseteq \texttt{double}\,x$, in which it would need to discover the lemma $\texttt{add}\,x\,(\texttt{Suc}\,x) \sqsubseteq \texttt{Suc}\,(\texttt{add}\,x\,x)$.

The theorem prover within Elea, described in Chapter 7, operates by rewriting properties into sufficient, and hopefully simpler, properties. A property has been proven if Elea is able to rewrite it to `tt` (truth). That Elea is essentially a property simplifier allowed the development of the fold discovery rewrite in Chapter 8, which is responsible for the most complex of Elea's potential rewrites, including the rewriting of $\texttt{sorted}\,(\texttt{isort}\,xs)$ to $\texttt{True}$. This rule uses Elea as a property simplifier within itself, in order to discover the shape of a required rewrite, and then uses Elea again as a theorem prover to check that this rewrite is sound.

Rather than attempting to reason about operational semantics directly, using denotational semantics made it very simple to prove Elea sound in Chapter 9, as soundness could be proven within the meta-language of domain theory. Proving the fusion steps within Elea sound within this meta-language required a novel method, truncation fusion, which relies on a syntactic extension to Elea's internal functional language, truncated fixed-points. Existing soundness methods would have been applicable had it not been for the fold discovery rewrite step, as it introduces a new fixed-point.

Elea currently has two main weaknesses, as evidenced by the results in Chapter 10. The first is that it is unable to reason about properties requiring lemmas which do not hold for non-total terms, even if the overall property being proven does hold. For example, the property $\texttt{even}\,(\texttt{add}\,n\,m) \equiv \texttt{even}\,(\texttt{add}\,m\,n)$ holds even for non-total values of n and m, but interally this proof requires the commutativity of $\texttt{add}$, which does not hold for non-total terms, so Elea is unable to prove this property. The second weakness is that Elea can sometimes rewrite terms to be too defined, and invalidate $\sqsubseteq$ properties in the middle of proving them. A potential solution to this, using a language extension called foldseq, was discussed in Section 10.5.

As a closing remark, I conjecture that proof by fixed-point promotion is complete with respect to cyclic proof for properties of denotational approximation. This is to say, I believe that for any cyclic proof of $A \sqsubseteq B$, there exists a corresponding fixed-point promoted form for the left-hand side of the approximation, viz. an F and $x_1...x_n$ along with a proof that $A \sqsubseteq \text{fix}\,(F)\,x_1...x_n$, as well as a proof of $F\;(\text{fn}\;x_1,...,x_n.\,B)\;x_1...x_n \sqsubseteq B$. There is a degenerate solution to this, which is to choose $F = \text{fn}\;f.\,A$ or $F = \text{fn}\;f.\,B$ and the set of $x_1...x_n$ to be empty, which corresponds to a cyclic proof which never actually uses its cyclic assumption. In addition, the method used in the original cyclic proof of $A \sqsubseteq B$, such as induction or coinduction, will correspond to the method which proves $A \sqsubseteq \text{fix}\,(F)\,x_1...x_n$. So, the fixed-point promotion technique used in Elea corresponds to cyclic proof by truncation induction, a method given in Lemma 2.5 on page 33, since this is the cyclic method which underlies the soundness proof of Elea's fixed-point promoted form producing rewrite rules.

Appendix A

Term definitions

This appendix defines the function term synonyms used within this thesis. A synonym name followed by a prime ($'$) denotes the body of a fixed-point. For every such fixed-point body $\texttt{name}'$ given here, the following two synonyms are also defined:

$$\texttt{name} \overset{\text{def}}{=} \text{fix}\,(\texttt{name}') \qquad\qquad \texttt{name}^a \overset{\text{def}}{=} \text{fix}^a\,(\texttt{name}')$$

For example, the $\texttt{add}'_y$ synonym defined below automatically gives:

$$\texttt{add}_y \overset{\text{def}}{=} \text{fix}\left(\texttt{add}'_y\right) \qquad\qquad \texttt{add}^a_y \overset{\text{def}}{=} \text{fix}^a\left(\texttt{add}'_y\right)$$

A.1 Functions on **Bool**

$$\texttt{not} \overset{\text{def}}{=} \text{fn}\ p.\ \text{if}\ p\ \text{then}\ \texttt{False}\ \text{else}\ \texttt{True}$$

$$\texttt{or} \overset{\text{def}}{=} \text{fn}\ p, q.\ \text{if}\ p\ \text{then}\ \texttt{True}\ \text{else}\ q$$

$$\texttt{and} \overset{\text{def}}{=} \text{fn}\ p, q.\ \text{if}\ p\ \text{then}\ q\ \text{else}\ \texttt{False}$$

A.2 Functions on **Nat**

```
$\texttt{add}' \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}), x, y.$
    case $x$ of
        $\texttt{0} \to y$
        $\texttt{Suc}\ x' \to \texttt{Suc}\ (f\ x'\ y)$
```

```
$\texttt{it-add}' \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}), x, y.$
    case $x$ of
        $\texttt{0} \to y$
        $\texttt{Suc}\ x' \to f\ x'\ (\texttt{Suc}\ y)$
```

```
$\texttt{add}'_y \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat}), x.$
  case $x$ of
    $\texttt{0} \to y$
    $\texttt{Suc}\, x' \to \texttt{Suc}\, (f\, x')$
```

```
$\texttt{double}' \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat}), x.$
  case $x$ of
    $\texttt{0} \to \texttt{0}$
    $\texttt{Suc}\, x' \to \texttt{Suc}\, (\texttt{Suc}\, (f\, x'))$
```

```
$\texttt{eq}' \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}), x, y.$
  case $x, y$ of
    $\texttt{0}, \texttt{0} \to \texttt{True}$
    $\texttt{0}, \texttt{Suc}\, y' \to \texttt{False}$
    $\texttt{Suc}\, x', \texttt{0} \to \texttt{False}$
    $\texttt{Suc}\, x', \texttt{Suc}\, y' \to f\, x'\, y'$
```

```
$\texttt{lq}' \overset{\text{def}}{=}$ fn $(f : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}), x, y.$
  case $x, y$ of
    $\texttt{0}, \texttt{0} \to \texttt{True}$
    $\texttt{0}, \texttt{Suc}\, y' \to \texttt{True}$
    $\texttt{Suc}\, x', \texttt{0} \to \texttt{False}$
    $\texttt{Suc}\, x', \texttt{Suc}\, y' \to f\, x'\, y'$
```

A.3 Functions on $\texttt{List}_\tau$

```
$\texttt{app}' \overset{\text{def}}{=}$ fn $(f : \texttt{List}_\tau \to \texttt{List}_\tau \to \texttt{List}_\tau), xs, ys.$
  case $xs$ of
    $\texttt{[ ]} \to ys$
    $x :: xs' \to x :: f\, xs'\, ys$
```

```
$\texttt{snoc}'_y \overset{\text{def}}{=}$ fn $(f : \texttt{List}_\tau \to \texttt{List}_\tau), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{[}y\texttt{]}$
    $x :: xs' \to x :: f\, xs'$
```

```
$\texttt{rev}' \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\tau \to \texttt{List}_\tau), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{[ ]}$
    $x :: xs' \to \texttt{snoc}_x\ (f\ xs')$
```

```
$\texttt{it-rev}' \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\tau \to \texttt{List}_\tau \to \texttt{List}_\tau), xs, ys.$
  case $xs$ of
    $\texttt{[ ]} \to ys$
    $x :: xs' \to f\ xs'\ (x :: ys)$
```

```
$\texttt{elem}' \overset{\text{def}}{=} \text{fn}\ (f : \texttt{Nat} \to \texttt{List}_{\texttt{Nat}} \to \texttt{Bool}), n, xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{False}$
    $x :: xs' \to \texttt{or}\ (\texttt{eq}\ n\ x)\ (f\ n\ xs')$
```

```
$\texttt{elem}'_n \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_{\texttt{Nat}} \to \texttt{Bool}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{False}$
    $x :: xs' \to \texttt{or}\ (\texttt{eq}\ n\ x)\ (f\ xs')$
```

```
$\texttt{elem-snoc}'_{n,y} \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_{\texttt{Nat}} \to \texttt{Bool}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{eq}\ n\ y$
    $x :: xs' \to \texttt{or}\ (\texttt{eq}\ n\ x)\ (f\ xs')$
```

```
$\texttt{filter}'_p \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\tau \to \texttt{List}_\tau), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{[ ]}$
    $x :: xs' \to \text{if}\ p\ x\ \text{then}\ x :: (f\ xs')\ \text{else}\ f\ xs'$
```

```
$\texttt{filter-snoc}'_{p,n} \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\tau \to \texttt{List}_\tau), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \text{if}\ p\ n\ \text{then}\ \texttt{[}n\texttt{]}\ \text{else}\ \texttt{[ ]}$
    $x :: xs' \to \text{if}\ p\ x\ \text{then}\ x :: (f\ xs')\ \text{else}\ f\ xs'$
```

```
$\texttt{map}'_f \overset{\text{def}}{=} \text{fn}\ (g : \texttt{List}_\tau \to \texttt{List}_\tau), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{[ ]}$
    $x :: xs' \to f\ x :: g\ xs'$
```

```
$\texttt{repeat}'_x \overset{\text{def}}{=} \text{fn}\ (xs : \texttt{List}_\tau).\ x :: xs$
```

```
$\texttt{iterate}'_f \overset{\text{def}}{=} \text{fn}\ (g : \tau \to \texttt{List}_\tau), x.\ x :: g\ (f\ x)$
```

```
$\texttt{sorted}' \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\texttt{Nat} \to \texttt{Bool}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{True}$
    $x :: xs' \to$
      case $xs'$ of
        $\texttt{[ ]} \to \texttt{True}$
        $x' :: xs'' \to \texttt{and}\ (\texttt{lq}\ x\ x')\ (f\ (x' :: xs''))$
```

```
$\texttt{insert}'_n \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\texttt{Nat} \to \texttt{List}_\texttt{Nat}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to [x]$
    $x :: xs' \to$ if $\texttt{lq}\ n\ x$ then $n :: xs$ else $x :: f\ xs'$
```

```
$\texttt{isort}' \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\texttt{Nat} \to \texttt{List}_\texttt{Nat}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to \texttt{[ ]}$
    $x :: xs' \to \texttt{insert}_x\ (f\ xs')$
```

```
$\texttt{sorted-insert}'_n \overset{\text{def}}{=} \text{fn}\ (f : \texttt{List}_\texttt{Nat} \to \texttt{Bool}), xs.$
  case $xs$ of
    $\texttt{[ ]} \to$        $\texttt{True}$
    $x :: xs' \to$    if $\texttt{lq}\ n\ x$
                  then $\texttt{sorted}\ (x :: xs')$
                  else
                    case $xs'$ of
                      $\texttt{[ ]} \to$         $\texttt{True}$
                      $x' :: xs'' \to$   if $\texttt{lq}\ n\ x'$
                                     then $f\ (x' :: xs'')$
                                     else $\texttt{and}\ (\texttt{lq}\ x\ x')\ (f\ (x' :: xs''))$
```

Bibliography

[1] Samson Abramsky and Achim Jung. Domain theory. Handbook of logic in computer science, 3:1–168, 1994.

[2] et al. Arnold, Ken. The Java programming language, volume 2. Addison-wesley, 2000.

[3] S. Baker. Aspects of the Constructive Omega Rule within Automated Deduction. PhD thesis, University of Edinburgh, 1992.

[4] S. Baker, A. Ireland, and A. Smaill. On the use of the constructive omega-rule within automated deduction. Logic Programming and Automated Reasoning: International Conference LPAR '92 St. Petersburg, Russia, July 15–20, 1992 Proceedings, pages 214–225, 1992.

[5] Siani Baker and Alan Smaill. A proof environment for arithmetic with the omega rule. Integrating Symbolic Mathematical Computation and Artificial Intelligence: Second International Conference, AISMC-2 Cambridge, United Kingdom, August 3–5, 1994 Selected Papers, pages 115–130, 1995.

[6] Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. ACM SIGPLAN Notices, 45(11):135–146, 2010.

[7] Maximilian C Bolingbroke. Call-by-need supercompilation. University of Cambridge, Computer Laboratory, Technical Report, UCAM-CL-TR-835, 2013.

[8] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In Automated Reasoning with Analytic Tableaux and Related Methods, volume 3702 of Lecture Notes in Computer Science, pages 78–92. Springer, 2005.

[9] Alan Bundy. The automation of proof by mathematical induction, volume 1 of Handbook of automated reasoning. Elsevier, 1999.

[10] Alan Bundy. Rippling: meta-level guidance for mathematical reasoning, volume 56 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[11] Alan Bundy, Mateja Jamnik, and Andrew Fugard. What is a proof? Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 363(1835):2377–2391, 2005.

[12] Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. Artificial intelligence, 62:185–253, 1993.

[13] Alan Bundy, Frank Van Harmelen, Jane Hesketh, Alan Smaill, and Andrew Stevens. A rational reconstruction and extension of recursion analysis. In IJCAI, pages 359–365, 1989.

[14] Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In 10th International Conference on Automated Deduction, volume 449 of Lecture Notes in Computer Science, pages 647–648. Springer, 1990.

[15] Alan Bundy, Frank Van Harmelen, Alan Smaill, and Andrew Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In 10th International Conference on Automated Deduction, volume 449 of Lecture Notes in Computer Science, pages 132–146. Springer, 1990.

[16] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. Journal of the ACM (JACM), 24:44–67, 1977.

[17] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. ACM SIGPLAN notices, 46(4):53–64, 2011.

[18] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. HipSpec: Automating Inductive Proofs of Program Properties. volume 7898 of Lecture Notes in Computer Science, pages 16–25. Springer, 2012.

[19] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In Tests and Proofs, pages 6–21. Springer, 2010.

[20] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. Mathematical systems theory, 13(1):131–180, 1979.

[21] Lucas Dixon and Jacques Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In Automated Deduction–CADE-19, volume 2741 of Lecture Notes in Computer Science, pages 279–283. Springer, 2003.

[22] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. New Generation Computing, 20:75–99, 2002.

[23] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. Theoretical Computer Science, 90:61–79, 1991.

[24] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? Electronic notes in theoretical computer science, 44:146–160, 2001.

[25] Geoff W Hamilton. Higher order deforestation. In Programming Languages: Implementations, Logics, and Programs, volume 1140 of Lecture Notes in Computer Science, pages 213–227. Springer, 1996.

[26] Geoff W Hamilton and Neil D Jones. Proving the correctness of unfold/fold program transformations using bisimulation. In Perspectives of Systems Informatics, pages 153–169. Springer, 2012.

[27] Geoffrey William Hamilton. Compile-time optimisation of store usage in lazy functional programs. PhD thesis, University of Stirling, 1993.

[28] Geoffrey William Hamilton. Distillation: extracting the essence of programs. In Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 61–70. ACM, 2007.

[29] Graham Hutton. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9:355–372, 1999.

[30] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. Journal of automated reasoning, 16:79–111, 1996.

[31] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. In Bulletin-European Association for Theoretical Computer Science, volume 62, pages 222–259, 1997.

[32] Moa Johansson, Lucas Dixon, and Alan Bundy. Interactive theorem proving: First international conference, itp 2010, edinburgh, uk, july 11-14, 2010. proceedings. volume 6172 of Lecture Notes in Computer Science, pages 291–306. Springer, 2010.

[33] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. Journal of Automated Reasoning, 47(3):251–289, 2011.

[34] Neil D Jones, Carsten K Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice Hall, 1993.

[35] Simon L Peyton Jones. Haskell 98 language and libraries: the revised report. Cambridge University Press, 2003.

[36] Peter A Jonsson and Johan Nordlander. Taming code explosion in supercompilation. In Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation, pages 33–42. ACM, 2011.

[37] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on, pages 23–34. IEEE, 1996.

[38] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. Computer-aided reasoning: an approach. Kluwer Academic Publishers, 2000.

[39] Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In Perspectives of Systems Informatics, volume 5947 of Lecture Notes in Computer Science, pages 193–205. Springer, 2010.

[40] Ilya Grigorievich Klyuchnikov. Supercompiler hosc 1.0: under the hood. Technical report, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2009.

[41] Ilya Grigorievich Klyuchnikov. Supercompiler hosc 1.1: proof of termination. Technical report, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2010.

[42] Ilya Grigorievich Klyuchnikov. Supercompiler hosc 1.5: homeomorphic embedding and generalization in a higher-order setting. Technical report, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2010.

[43] Ilya Grigorievich Klyuchnikov. Supercompiler hosc: proof of correctness. Technical report, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2010.

[44] Laurent Kott. About transformation system: A theoretical study. Program transformations, pages 232–247, 1978.

[45] K Rustan M Leino. Automating induction with an SMT solver. In Verification, Model Checking, and Abstract Interpretation, volume 5947 of Lecture Notes in Computer Science, pages 315–331. Springer, 2012.

[46] K Rustan M Leino and Michał Moskal. Co-induction simply. In FM 2014: Formal Methods, pages 382–398. Springer, 2014.

[47] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Static Analysis, volume 1503 of Lecture Notes in Computer Science, pages 230–245. Springer, 1998.

[48] Dorel Lucanu, Eugen-Ioan Goriac, Georgiana Caltais, and Grigore Roşu. Circ: A behavioral verification tool based on circular coinduction. In Algebra and Coalgebra in Computer Science, pages 433–442. Springer, 2009.

[49] Dorel Lucanu and Grigore Roşu. Circ: A circular coinductive prover. In Algebra and Coalgebra in Computer Science, volume 5728 of Lecture Notes in Computer Science, pages 372–378. Springer, 2007.

[50] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. Communications of the ACM, 16:491–502, 1973.

[51] Simon David Marlow. Deforestation for higher-order functional programs. PhD thesis, University of Glasgow, 1995.

[52] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In Functional Programming Languages and Computer Architecture, volume 523 of Lecture Notes in Computer Science, pages 124–144. Springer, 1991.

[53] James H Morris Jr. Another recursion induction principle. Communications of the ACM, 14:351–354, 1971.

[54] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In ACM SIGPLAN Notices, volume 42, pages 143–154. ACM, 2007.

[55] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In ACM SIGPLAN Notices, volume 42, pages 143–154. ACM, 2007.

[56] Y Onoue, Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. A calculational fusion system HYLO. In Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi, pages 76–106. Chapman & Hall, Ltd., 1997.

[57] Lawrence Paulson. Deriving structural induction in LCF. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, Semantics of Data Types, volume 173 of Lecture Notes in Computer Science, pages 197–214. Springer, 1984.

[58] Gordon D. Plotkin. LCF considered as a programming language. Theoretical computer science, 5(3):223–255, 1977.

[59] Grigore Roşu and Dorel Lucanu. Circular coinduction: A proof theoretical foundation. In Algebra and Coalgebra in Computer Science, pages 127–144. Springer, 2009.

[60] Ben Rudiak-Gould, Alan Mycroft, and Simon Peyton Jones. Haskell is not not ml. In Programming Languages and Systems, pages 38–53. Springer, 2006.

[61] David Sands. Total correctness by local improvement in program transformation. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 221–232. ACM, 1995.

[62] David Sands. Proving the correctness of recursion-based automatic program transformations. Theoretical Computer Science, 167:193–233, 1996.

[63] David Sands. Total correctness by local improvement in the transformation of functional programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 18:175–234, 1996.

[64] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In Tools and Algorithms for the Construction and Analysis of Systems, pages 407–421. Springer, 2012.

[65] Morten H. Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In Proceedings of ILPS'95, the International Logic Programming Symposium, pages 465–479. MIT Press, 1995.

[66] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. Journal of Functional Programming, 6:811–838, 1996.

[67] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In ACM SIGPLAN Notices, volume 37, pages 124–132. ACM, 2002.

[68] Valentin F. Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, 8:292–325, 1986.

[69] Valentin F. Turchin. The algorithm of generalization in the supercompiler. Partial Evaluation and Mixed Computation, 531:549, 1988.

[70] Irene Lobo Valbuena and Moa Johansson. Conditional Lemma Discovery and Recursion Induction in Hipster. In 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015), volume 72, 2015.

[71] Guido van Rossum and Fred L. Drake. The python language reference. Technical report, Python software foundation, Amsterdam, Netherlands, 2010.

[72] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In ESOP'88, pages 344–358. Springer, 1988.

[73] CHIN Wei-Ngan. Safe fusion of functional expressions. In Proceedings of the 1992 ACM Conference on LISP and Functional Programming, page 11. Pearson Education, 1992.