University of Kent

Kent Academic Repository

Wang, Meng (2011) *Bidirectional Programming and its Applications.* Doctor of Philosophy (PhD) thesis, University of Oxford.

Downloaded from

https://kar.kent.ac.uk/55795/ The University of Kent's Academic Repository KAR

The version of record is available from

This document version

Other

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

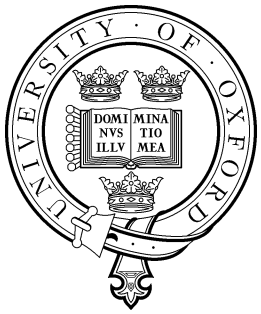
Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Bidirectional Programming and its Applications



Meng Wang

Wolfson College

University of Oxford

Submitted for the degree of Doctor of Philosophy

Trinity 2010

Bidirectional Programming and its Applications

Meng Wang

Wolfson College

University of Oxford

Submitted for the degree of Doctor of Philosophy

Trinity 2010

Abstract

Many problems in programming involve pairs of computations that cancel out each other's effects; some examples include parsing/printing, embedding/projection, marshalling/unmarshalling, compressing/de-compressing etc. To avoid duplication of effort, the paradigm of bidirectional programming aims at to allow the programmer to write a single program that expresses both computations. Despite being a promising idea, existing studies mainly focus on the view-update problem in databases and its variants; and the impact of bidirectional programming has not reached the wider community. The goal of this thesis is to demonstrate, through concrete language designs and case studies, the relevance of bidirectional programming, in areas of computer science that have not been previously explored.

In this thesis, we will argue for the importance of bidirectional programming in programming language design and compiler implementation. As evidence for this, we will propose a technique for incremental refactoring, which relies for its correctness on a bidirectional language and its properties, and devise a framework for implementing program transformations, with bidirectional properties that allow program analyses to be carried out in the transformed program, and have the results reported in the source program.

Our applications of bidirectional programming to new areas bring up fresh challenges. This thesis also reflects on the challenges, and studies their impact to the design of bidirectional systems. We will review various design goals, including expressiveness, robustness, updatability, efficiency and easy of use, and show how certain choices, especially regarding updatability, can have significant influence on the effectiveness of bidirectional systems.

Statement of Originality

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of the author's knowledge and belief, the thesis contains no material previously published or written by another person except where due references are made.

Chapter 3 of the thesis has been published [WGMH10]. The author claims originality of the idea behind the work, and played the leading role in the development of the paper. Kazutaka Matsuda was involved in the discussion of the language design and contributed to improving the presentation of the paper. An implementation of the system (not included as a contribution in this thesis) is due to him. Jeremy Gibbons and Hu Zhenjiang oversaw the project and provided many suggestions for improvement.

Acknowledgements

I would like to express my deepest gratitude to Jeremy Gibbons, who has given me the best support I can ever hope for from a supervisor. He has been constantly inspiring, encouraging and approachable. I benefited tremendously from his insightful and frank comments; and in him I see a perfect role model of a good scientist, both in work and in life. I cherish every moment of our meetings; let it be in the lab, by a barbecue pit, or in the church. If I am in any way a better researcher today than a few years ago, it is all due to him.

My special thanks also go to Ralf Hinze, who is always exceptionally quick in thinking, and yet patient in explaining. His passion for research is infectious. Being able to sit next door to some of the world's best computer scientists like him is truly a great privilege.

During my D.Phil study, I had the chance of visiting Japan National Institute of Informatics and University of Tokyo as an intern, under the supervision of Hu Zhenjiang. It was an eye opening experience in every aspect. The effectiveness of the team work, lead by Prof. Hu, and the hospitality of the group members made my stay very much enjoyable.

A large number of my lunch breaks in Oxford was spent with Bruno Oliveira, a true friend and a close colleague. It was such a pleasant experience to work with him. Kazutaka Matsuda and Janis Voigtländer are another two great researchers, who I had the honour of working and befriending with. The wonderful memory of the time spend with them will never go away.

The Algebra of Programming group in Oxford have given me the perfect research support; the lively regular group meetings are definitely a source of inspiration. Many members in the group, in particular Richard Bird, Geraint Jones and Nick Wu, have offered much useful advice to my work. I would like to thank all of them.

Last but not least, I need to thank EPSRC for its generous financial support. Without it, all this would not have happened.

Contents

1 Introduction **1**

1.1 Reversible Programming . . . 2
1.2 The View-Update Problem . . . 4
1.3 Bidirectional Programming for the Masses . . . 5
1.3.1 A Motivating Scenario . . . 5
1.4 Outline of the Thesis . . . 6

2 Related Work and Preliminaries **9**

2.1 Related Work . . . 10
2.1.1 Bidirectional Languages . . . 10
2.1.1.1 INV and X . . . 10
2.1.1.2 Lenses . . . 11
2.1.1.3 Point-free Lenses . . . 12
2.1.1.4 Constraint Maintainers . . . 12
2.1.1.5 QVT Relational . . . 13
2.1.1.6 2LT and Data Refinement . . . 13
2.1.2 Bidirectionalization . . . 14
2.1.2.1 Syntactic Bidirectionalization . . . 14
2.1.2.2 Semantic Bidirectionalization . . . 15
2.2 Preliminaries . . . 16
2.2.1 Point-free Programming . . . 17
2.2.2 Bidirectional Properties . . . 19

3 Looking from the Right **21**

3.1 Introduction . . . 22
3.1.1 Program Development . . . 22
3.1.2 Pattern Matching . . . 22

3.2 Data Abstraction . . . 24
3.2.1 An Example . . . 25
3.3 The Right-Invertible Language 'RINV' . . . 28
3.3.1 The Primitive Functions . . . 30
3.3.2 The Constructors . . . 31
3.3.3 The Combinators . . . 32
3.3.3.1 Composition, Sum and Product . . . 32
3.3.3.2 Recursion . . . 34
3.3.4 Programming in RINV . . . 35
3.4 Selective Refactoring . . . 37
3.4.1 The Computation Law . . . 38
3.4.2 Refactoring by Translation . . . 40
3.4.3 Optimization . . . 41
3.4.4 More Examples . . . 43
3.4.4.1 Join Lists . . . 43
3.4.4.2 Binary Numbers . . . 44
3.4.4.3 Sized Trees . . . 45
3.5 Discussion . . . 45
3.5.1 RINV Expressiveness . . . 45
3.5.2 The Dual Story . . . 46
3.5.3 Nested and Overlapping Patterns . . . 47
3.5.4 Right-invertible vs. Bidirectional . . . 48
3.6 Further Applications . . . 49
3.6.1 Pattern Matching for ADTs . . . 49
3.6.2 Stream Fusion . . . 50
3.7 Related Work . . . 51
3.8 Conclusion . . . 54

4 Looking from the Left **55**
4.1 Introduction . . . 56
4.2 Bidirectionalization Transformation . . . 58
4.2.1 An Example . . . 59
4.3 The Algorithm . . . 62
4.3.1 Constructing the Complement Function . . . 62
4.3.2 Tupling . . . 65

4.3.3 Inverting the Tupled Function 66
4.3.3.1 Multiple Use of Variables 66
4.3.4 Properties of Bidirectionalization 67
4.3.5 Projection of Data 68
4.3.6 The Anatomy of Complements 69
4.4 Program Analysis Result Reporting 70
4.4.1 Annotated Expressions 71
4.4.2 Bidirectionalizing Transformations 71
4.4.2.1 Rearranging expressions 71
4.4.2.2 Removing expressions 74
4.4.2.3 Creating expressions 76
4.4.2.4 Fresh Variables 78
4.5 Discussion 79
4.5.1 Reuse of Existing Code 80
4.5.2 Syntactic vs. Semantic Bidirectionalization 81

5 Looking from the Top **83**
5.1 Introduction 84
5.1.1 A Small Example 85
5.2 The Overall Setting 87
5.2.1 Tree Navigation 87
5.2.2 Local Editing 90
5.2.3 Changed-based Bidirectional Systems 91
5.3 Locality Preservation 92
5.3.1 Alignment 92
5.3.2 Exploiting Regularity 95
5.4 Change-Based Putback Functions 102
5.4.1 Indexing and Reflecting 102
5.4.2 The Change-Based Putback Function 104
5.5 More Refined Locality 108
5.6 Discussion 109
5.6.1 Context-Sensitive Editing 109
5.6.2 Connection with Other Bidirectional Approaches 110

6 Conclusion **113**

6.1 Summary 114

6.2 Totality of Putback Functions 116

6.3 Combinator-Based Bidirectional Languages 116

6.4 Bidirectionalization 117

Bibliography **118**

List of Figures

5.1 The Context-Focus Representation 87
5.2 Source-View Alignment 93
5.3 Operation-based Putback 105

Chapter 1

Introduction

Those behind cried "Forward!"
And those before cried "Back!"
(T.B. Macaulay, Lays of Ancient Rome – Horatius (1842))

At the beginning of the computing era, the forefathers of computer science made the decision to build machines supporting deterministic uni-directional computation, a legacy that lingers till today. As a result, computer programs constantly perform operations that are forgetful about their execution history, leaving the machine in a state whose immediate predecessor is ambiguous. Nevertheless, the need to reverse a computation does arise in many contexts – database recovery, undoing edits, debugging, etc.

The exploration of a different design began in hardware, when it was recognized that all microscopic physical processes are inherently conservative, and the erasure of information consumes much energy. In response to this finding, reversible Turing machines were developed [Ben73] and conservative chips that do not erase information were built [You94].

Yet, even with all hardware circuits re-implemented with conservative logic, we are still facing problems at the highest level of computer usage where the traditional computation model is information-lossy. At this level, the interest has gone beyond energy saving to more structured programming practise. For decades, the need for invertible programming has been dealt with using brute force. Generations of programmers have written parser/printer, embedding/projection, marshalling/unmarshalling, compressing/decompressing, etc. as separate programs. This is a significant duplication of work, because the two programs in a pair are closely related. This close relation introduces a maintenance problem, besides being error-prone; the pair have to be consistent to avoid bugs, and changes to one entail matching changes to the other.

The way we tackle this problem (along with many others), is by designing languages that execute bidirectionally. In such languages, any program from type A to type B (the *retrieval function*) is always coupled with an automatically generated "backwards" version (the *putback function*), whose correctness with regards to certain invertibility properties is guaranteed. Along with the development of bidirectional languages, new applications of them emerge, which, in turn, promotes new language designs.

1.1 Reversible Programming

A direct approach to make programs bidirectional is to make every construct of the language two-way deterministic. Following this principle, the paradigm of reversible programming is born, with the design of *Janus* (firstly proposed in [Lut86] but later published in [YAG08]) being most notable. Janus is an imperative language with reversible control structures and assignments. For example, in addition to an entry condition at

the beginning of any branch, there is an exit assertion whose value distinguishes the different branches. The exit assertion will be used as an entry condition in the reverse direction, and vice versa. An assignment in Janus is tagged with the operation that updates the value, and only injective operations are allowed. In this way, programmers always think bidirectionally while programming. Janus is fully symmetric; there is no distinction between the two directions of execution.

The symmetry of reversible programming allows elegant specification of its properties. If we use $f :: A \to B$ and $f^\circ :: B \to A$ to denoted the retrieval and putback functions, we have invertibility specified as program inversion: $f \circ f^\circ = id_B$ and $f^\circ \circ f = id_A$, where $\circ$ stands for function composition and id_X is the identity function over type X. As well as being elegant, this symmetry also places a cap on the expressiveness of the reversible approach: f must be bijective. Modulo information encoded in f itself, its input and output of it necessarily contain exactly the same information, probably with different presentations. This is certainly inadequate. Quoting from [Ste10], "*More generally, bijective transformations are the exception rather than the rule: the fact that one model contains information not represented in the other is part of the reason for having separate models.*"

To circumvent this restriction, in a later implementation of a functional reversible language [MHT04b], the totality requirement of invertibility is given a different interpretation. Instead of having id over all the values of type A or B, the new invertibility concerns only the exact domain and range of f, which produces

$$f \circ f^\circ = id_{range(f)}$$
$$f^\circ \circ f = id_{domain(f)}$$

or

$$f^\circ \circ f \circ f^\circ = f^\circ$$
$$f \circ f^\circ \circ f = f$$

As a result, the language in [MHT04b] is better-off and handles duplication, which was previously out of reach. Still, this rules out the majority of functions we would like to define, due the fact that f has to be injectivity to satisfy either the above sets of law.

1.2 The View-Update Problem

Another driving force of bidirectional programming is from the database community, with the so called *view-update problem* being a subject of study for decades [DB82, BS81, GPZ88]. The main goal is to map updates of a view back to the original source data "properly". Linguistic solutions to the problem have been a hot research topic recently [FGM+07, MHN+07, BFP+08, FPP08, Voi09, FPZ09]. In this setting, the abstract nature of views is fundamentally at odds with the injective-then-invertible principle. New notations are invented and different sets of rules are proposed. Notably, a framework for bidirectional systems where (all or part of) the original source is copied and is used in the putback computation has been developed. As a result, the symmetry between the two directions of computation is broken: a retrieval function (also known as *get*) has type $S \to V$ from source to view, while a putback function (also known as *put*) has type $(V, S) \to S$. Effectively, the retrieval function is "injectivized" by the copying of the input, intended for a deterministic putback execution. Accordingly, the invertibility property in reversible programming is transformed into the following two *definitional properties* [CFH+09, Ste07]:

Consistency $get\ (put\ (v, s)) = v$

Acceptability $put\ (get\ s, s) = s$

Consistency (also known as the PutGet law) roughly corresponds to right-invertibility, which basically ensures that all updates on a view are captured by the updated source, and *acceptability* (also known as the GetPut law) roughly corresponds to left-invertibility, prohibiting changes to the source if no update has been done to the view. Bidirectional systems satisfying the above two laws are sometimes called *well-behaved* [FGM+07]. There is an optional *undoability* property (also known as the PutPut law) : $put\ (v', put\ (v, s)) = put\ v'\ s$. Additional *quality properties* such as *incrementality*, *minimality of changes*, and *preservation of recent changes* are desirable, but are not well understood formally [CFH+09].

Despite the fact that with the copied source injectivity is assured, it is never trivial to merge the changes in the view into the source in a law-abiding way, a situation made worse by the popular assumption that updates are arbitrary and only the resulting updated view, not the process by which this is obtained, is known. Consider a simple example of concatenating two lists. If the updated view is one element short, it is not clear whether an element shall be removed from the first or the second list, or one of many other possible arrangements.

Another self-limiting trend in the development of bidirectional languages is that the application has been more or less restricted to a small number of areas, in particular to the view-update of XML structures. Its relevance to a wider community has not been fully explored.

1.3 Bidirectional Programming for the Masses

We think the applicability of bidirectional programming is underestimated. There are many high-level programming problems other than database updates that require some sort of invertibility. As an example, let us consider type checking and error reporting.

1.3.1 A Motivating Scenario

Anyone who had the experience of writing research papers about type systems and implementing the ideas into real languages knows well that the two are very different things, even with a functional implementation where the structure of the program is very similar to the typing rules. The real problem is the explosion of the number of cases to cover all features of a realistic language. Since all the syntactic sugar is translated into a small core language anyway, why can't we just type check the core?

It is done the hard way for a very good reason: type error reporting. We have to report any error in the "language" of the source code, including the exact location and context, not in the "language" of some unrecognisable internal code. This need basically fixes type checking to the first stage of compilation, before any desugaring or program transformation comes into the picture – unless the transformations are bidirectional! It is probably difficult to persuade compiler writers to implement a "sugar" or a "deoptimize" function for every transformation they write. A bidirectional language is a better bet. However, if we shop around, it is likely that the existing systems are not expressive enough: either because over-demanding bidirectional rules need to be satisfied, or because the updates are assumed to be arbitrary, which both necessarily limit the expressiveness of the language. In our case here, neither assumption is true: for type error reporting, only acceptability is needed (because users will not inspect the AST), and type checking will only indicate errors but not alter the program structure. At the same time, language expressiveness and user friendliness are crucial here; it will be very difficult, if not outright impossible, to implement a desugar or optimization procedure in a stylized combinator library, restricting oneself to injective operations.

Similar mismatches are common when we step out of the traditional comfort zone of database view updates. This thesis is dedicated to extending bidirectional programming to a broader frontier. It turns out that there is no golden law that applies to all domains in this field. Quoting from [FGM+07] on how bidirectional languages shall be designed, "*To reconcile this tension* [between language expressiveness and robustness]*, any approach to the ... problem must be carefully designed with a particular application domain in mind.*". In this thesis, as expected, new solutions have to be developed and new languages are designed. Like many others, we hope to see further by standing on the shoulders of giants. As a result, though with very different styles, the languages proposed in this thesis are never too exotic for ordinary programmers, and at the same time well fitted for the intended applications. The main technical contributions of this thesis cover a rather large spectrum, but all centre around a common theme: Bidirectional techniques are widely applicable and the update information shall be a primary consideration in design.

1.4 Outline of the Thesis

We have set ourselves the challenge of looking at bidirectional programming from a non-conventional perspective. Before setting out to tackle it, we will firstly discuss the state of the art in Chapter 2, by reviewing existing work in the field of bidirectional programming. There is no advanced prerequisite for this thesis, other than some familiarity with the programming language Haskell. Nevertheless, we include a short preliminary section in Chapter 2 for the purpose of standardising the terminologies that is going to be used throughout the thesis.

The main technical chapters include Chapter 3, 4 and 5. In Chapter 3, we start with a small point-free combinator library, which is right-invertible, for the purpose of supporting incremental refactoring of programs written with pattern matching. Based on the library, we will design a translation mechanism for program refactoring. The resulting system supports proper abstraction (as found in smooth refactoring), as well as sound and elegant equational reasoning (a main thrust of pattern matching). We will also demonstrate additional use scenarios of our proposal, involving pattern matching for selected abstract datatypes and fusion for streams.

Chapter 4 follows the same development style by designing a bidirectional system for a specific type of applications. However, the similarity between the two chapters does not go beyond the organization level. The bidirectional technique used in Chapter 4 syntactically rewrites a uni-directional program, in a Haskell-like language, by generating

its "backwards" counterpart; the language is pointwise in style and is very expressive, a design that is dictated by the intended application. The idea is to be able to "bidirectionalize" program transformations, so that program analyses performed on the transformed program can have their results mapped back to the source program, which is recognizable to the user. We will showcase the effectiveness of the proposal by studying common program transformation scenarios as examples. In this chapter, we will solve the motivating problem described earlier on.

Chapter 5 departs from the development pattern of the previous two chapters, by looking at a general mechanism for improving run-time performance of bidirectional systems. The idea is that a putback function will reconstruct only the part of the source that is affected by a change in the view, not the whole of it. Motivated by improving efficiency, we will resort to a design that demands more information about a change to a view, in addition to the result of such a change, which saves the effort of performing a difference analysis on the two view values, and allows us to work only on the affected part; this will improve the performance of the putback function if a small change in the view corresponds to a small change to the source, a property that clearly does not hold for arbitrary retrieval functions. We will identify some sufficient semantic conditions, which can be used to select retrieval functions that could benefit from this optimization, and prove the correctness of our approach with respect to the standard bidirectional laws.

We then summarize in Chapter 6 and discuss future directions.

Chapter 2

Related Work and Preliminaries

2.1 Related Work

The need of functionalities that has the ability to "cancel" the effect of certain operations is common in hard-/soft-ware systems, ranging from hardware design to model-driven software architecture, and from database updates to program analysis. Due to this diversity, solutions developed in different disciplines are largely incomparable under different settings. For the purpose of this thesis, which approaches the problem from a "programming" perspective, we will survey the bidirectional systems that has some flavour of programming languages in them. Related work that concerns only specific chapters will be discussed in the respective chapters.

2.1.1 Bidirectional Languages

2.1.1.1 INV and X

The obvious bidirectional languages are the ones with injective computations, whose inverses are readily available. Notably in INV [MHT04b], which is a combinator library, only injective functions are syntactically allowed. The most novel operator of INV is *dup* f, which duplicates the input and applies f to one copy. In the inverse direction, the two copies of the duplicated input are checked for consistency before being restored. The language INV has all properties that one expects from a refined invertible language: elegant invertibility and limited expressiveness. The authors went on to extend the language with non-injective operations, giving rise to X [MHT04a, HMT04]. Given information losses, the operations in X must copy the original input and use them on the way back. The ingenuity of X is that all the messy bookkeeping of discarded information is handled in the background instead of being visible to the programmers, particularly through an embedding process into INV. Effectively, programs in X enjoy the same kind of invertibility property, but with extended expressiveness. To deal with non-surjectivity, especially non-surjectivity introduced by the *dup* operation, an alternative semantics is introduced, which tags views to include information such as active editing and insertion/deletion of elements in a structure. When two values come to an inverted *dup* operation, the edited one takes precedence; and knowing whether certain elements are newly inserted or deleted helps to align two duplicated structures that do not agree.

2.1.1.2 Lenses

The *lens* family of languages [FGM+07, BFP+08, FPP08, FPZ09, BCF+10a, BCF+10b] is designed originally for view-update of XML data, and is extended to handle various kinds of ad-hoc data as well. The putback function (known as *put*) takes both an updated view and the original source as inputs. Thus, lenses are not limited by the injective requirement. Yet, invertibility has to be specified with the less elegant PutGet and GetPut laws. Being a combinator-based language, user-defined lenses can be constructed by lens combinators from primitive lenses, and are guaranteed to be well behaved. One important primitive lens is *const*, where the copied original source is actually used by the putback function *put*: in the retrieval direction *const* maps any source to a constant value and in the putback direction the input view is ignored and the original source is restored. When combined with conditional and fixed-point combinators, non-trivial data projections, such as a filtering function, which selectively removes elements from a structure, can be expressed. In contrast to X, lenses do not attempt to hide the original source as an input of the putback function. On the contrary, there are combinators that require programmers to think bidirectionally and make explicit use of the additional input. For example, some of the conditional combinators branch in the putback direction not only based on the view input, but also the source input.

Instead of remaining partial, the lens framework supplements the partial *put* with a function *create* of type $V \to S$ that executes at the points where *put* fails, which effectively serves as a catch-all case. The function *create* is the right-inverse of *get*: it satisfies $get\ (create\ v) = v$, known as the CreateGet law. Though not unique, total *create* functions (right-inverses) always exist for surjective *get*s, and combining them with partial *put*s deliver total putback functions.

The surjectivity of *get*s is guaranteed by the use of *semantic* types [FCB08] to give precise bounds to the ranges. As a result, the types are very often complicated and do not respect phase distinction. For example, consider the function *const* v that maps any input to a constant v; its type is written as $A \to \{\, v \,\}$, where the value v is lifted to the type level. (This example is slightly simplified from [FGM+07], but the same principle applies.)

The framework has subsequently been extended with more fine grained indexing of data components (namely *chunks*) to properly deal with ordered string data [BFP+08]. A concept of *dictionary lenses* is introduced to maintain associations between unique indexes with individual chunks, which are not vulnerable to reordering and preserve *alignments* [BCF+10b]. The semantic foundation of lenses can be adapted by replacing

the strict equality in the laws to either user-defined *quotient equivalence* [FPP08] or *endorsed equivalence* [FPZ09] that ignores non-essential differences or only enforcing the part with high integrity respectively.

2.1.1.3 Point-free Lenses

Lenses are rather stylized, in a sense that programming using them is different from using mainstream general-purpose languages. In [PC10], the standard combinators of point-free programming are "lensified" to produce a bidirectional language that has a familiar feel. The design follows the bidirectional requirement of lenses faithfully. Though less expressive, the language is expected to be friendlier to program and calculate with. Coincidentally, point-free lenses are very similar in spirit to the right-invertible language we use in Chapter 3.

2.1.1.4 Constraint Maintainers

Tracing back to the more distant past, *constraint maintainers* [Mee98] closely resemble a bidirectional system for view updates. Given two objects of type A and B, a *maintainer*, as a pair of functions $\triangleright :: (A, B) \to B$ and $\triangleleft :: (A, B) \to A$, keeps some relation between them intact. In a sense, maintainers, being symmetric, are more general than lenses, by permitting changes in either party of two connected by a relation R, and coevolving the other party. The correctness of a maintainer is specified with respect to the relation.

Establish-L $(x \triangleleft y)\ R\ y$

Establish-R $x\ R\ (x \triangleright y)$

Skip-L $(x \triangleleft y) = x \Longleftarrow x\ R\ y$

Skip-R $(x \triangleright y) = y \Longleftarrow x\ R\ y$

The establishment rules ensure consistency of the relation after executing the maintainer; the skip rules avoid unnecessary changes if the two sides are already correctly related.

Maintainers are used for a wide range of relations including those for sets and numbers, which are not commonly touched by other bidirectional system. Given the multi-valued nature of relations, correct maintainers with respect to a relation are usually not unique. If orders are given to the effects of the maintainers, for example minimality of changes, *selectors* can be used to choose an optimal one.

When applied to structured data such as lists, the result of a change is not sufficient to establish the path of the change. In contrast to most other systems, maintainers express updates in term of editing scripts instead of the resulting value, which allows a more precise account of the updates. For example, on deleting an element from a list, the maintainer can be fed with information on the exact location of the deleted element, rather than simply a new list that is one element short. It is no surprise that the more that is known about an update, the better it can be mapped back to the source. A less explored, but a very important, advantage of looking into edit scripts is the potential of scalability: putback functions now process an edit operation in time proportional to the size of the change, not to the size of the structure. We will see development in this direction in Chapter 5.

2.1.1.5 QVT Relational

The value of bidirectional transformation is also recognized in the area of Model-Driven Development [Gro05], as "*in practice it will be necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions.*" [Ste10]. The Object Management Group (OMG) responded to this need by including a bidirectional language in the Query View Transformation (QVT) standard, namely QVT Relational [Gro05]. The language allows the specification of relations with enabling expressions (*when* clauses) and enforced expressions (*where* clauses). When the enabling expressions evaluate to true, the enforced expressions must also be true.

The semantics of QVT Relational is further specified with a "check then enforce" principle, which basically forbids the running of a transformation to change models that are already consistent. This semantics closely resembles the skip rules of maintainers we have seen above.

2.1.1.6 2LT and Data Refinement

Not all bidirectional languages have their design centred around handling updates on the value level. One that is not is the 2LT (Two-Level-Transformation) system [BCF+], which has a bidirectional combinator library as the core. Instead of addressing changes to individual values, the update is expressed as a format (type) evolution, which migrates a whole database to one in a new format. A typical usage of the system involves retrieving relational data into one in a hierarchical format, updating by enriching the hierarchical format, and putting back the updated data into a new relational format reflecting the

enrichment. The whole process is specified in terms of formats; and the corresponding value-level transformations are induced. The 2LT system can be implemented in a type preserving way [COV06], thanks to the rich type system of Haskell.

The 2LT system is an instance of the more general *calculational data refinement* [MG90, Oli08], where inequalities between two datatypes are witnessed by a pair of conversion functions α and γ. In contrast to isomorphism, the inequalities only requires $\gamma \circ \alpha = id$, but not the other way round. This is the dual of most other bidirectional systems, which usually go from a data-rich source to a data-poor view.

2.1.2 Bidirectionalization

Another solution other than outright implementations of bidirectional language is to *bidirectionalize* existing uni-directional programs. Such a process is termed *bidirectionalization* [MHN+07]. Basically there are two different approaches to the problem in the literature, that either *syntactically* rewrite a given program or *semantically* wrap it up without inspecting its code. We discuss both in detail in the following.

2.1.2.1 Syntactic Bidirectionalization

One way to bidirectionalize a uni-directional program is to make it injective. One can assume that the source s can be factored into the view v and its (fairly orthogonal) complement c in such a way that the pair (v, c) uniquely determines s. When v is updated, the complement c stays constant; if the updated result (v', c) remains in the range of the retrieval transformation, then the putback transformation can be constructed through program inversion. The constant nature of the complement gives rise to the name *constant complement* approach. It was firstly proposed in database research [BS81] and is later used in bidirectional programming [MHN+07].

As a very simple example, consider the natural number addition function $add\ (a, b) = a + b$; a complement value such as the first element a of the source together with the view $a+b$ is sufficient to recover the complete source. Putback transformations generated this way are always very well behaved, assuming successful execution. The challenge is to avoid potential conflicts between changeable views and constant complements. For example, updating the value of $a + b$ to one that is smaller than a conflicts with the complement value. Note that this does not mean that the updated view is not producible from any source, but the updated view is not producible from any source with the given complement. In this case, it is no longer straightforward to reflect the updates into the

source properly. Consequently, the putback functions produced are generally not total (in the sense that a putback function is case-exhaustive over the actual range of the retrieval function [FGM+07]); not all updated views that in theory have corresponding sources can be mapped back.

Since the complement value determines the updatability of the view, it is essential to minimize the overlap between the two. For a given source and retrieval function, the complement value is typically not unique: for example, the value a or the complete input (a, b) in the above example are both legitimate complements. However, it is obvious that the latter overlaps more with the view, and therefore is considered *bigger*, implying reduced updatability. Much effort has been made [MHN+07] for reducing derived complements. Despite the effort, it still turns into an update checker that rejects complement-violating updates to ensure the safety of putback executions. To make the check decidable, the language is restricted to be *affine* (no duplication of bound variables), and *treeless* (no nested function calls).

2.1.2.2 Semantic Bidirectionalization

A syntactical system needs to access the source code of retrieval functions, which can be unavailable at times. An alternative is *semantic* bidirectionalization [Voi09], which does not inspect the syntactic definitions. The system instead decomposes a source into atomic elements and their structural container; the elements are indexed into a dictionary and the structure skeleton decorated by the indices is fed to the retrieval function to produce a map of the transformation. View updates, expressible in terms of overwriting the corresponding entries in the dictionary, are eventually incorporated by reconstructing a source from the skeleton of indices and the updated dictionary. The equivalence of running retrieval functions on element- and index- decorated skeletons can be guaranteed by *free theorems* [Wad89], but only for parametrically polymorphic functions. Nevertheless, the more notable restriction is the prohibition of any update that alters the structure of the source, which implicitly serves as the constant complement here.

It is an interesting observation in [VHMW10] that combining the semantic and syntactic approaches actually helps in the task of reducing complements, because the element part of the complement, handleable by the semantic approach, often gets in the way of collapsing the structure part of the complement. By specializing the syntactic approach to structure only and leaving the elements to be dealt with semantically improves updatability of either system, though only in the intersection of their expressiveness domain.

It is also made apparent in the same article that as we go further down the path of reducing complements, the price for improvement, in term of complexity of the system and restriction to the retrieval functions, shoots up. The Pareto Principle suggests that it might be more profitable to look elsewhere.

2.2 Preliminaries

Haskell [PJ03] is the carrier language of this thesis; we assume basic understanding of Haskell with its standard features such as pattern matching, type classes, list comprehensions, etc. Throughout this thesis, we work in the setting of total functions; though for convenience, we don't always (in particular in Chapter 4, when we discuss program transformations) write down all cases of an input in a function definition, with the understanding that this practice does not compromise our claim, as "partial" functions of that sort can be easily turned into total functions by wrapping the outputs in *Maybe* type. A similar principle applies to certain putback functions that only construct valid sources for restricted updates on views; totality of these functions can be recovered by injecting the output into *Maybe* as well. However, it is obvious that only the non-*Nothing* outputs are interesting. So we refer the domain of a putback function as the set of inputs that leads to non-*Nothing* outputs.

In Haskell, there are two basic styles of function definition, namely the *pointwise* style and the *point-free* style. We have already seen examples of both in previous sections. For example, in the specification of laws for lenses in Section 2.1.1.2

$$get\ (put\ (v, s)) = v$$

and

$$put\ (get\ s, s) = s,$$

we used the pointwise style, where the definitions are described by function application to arguments. In the point-free style, definitions are described exclusively in terms of higher-order combinators, like function compositions: for example the definition of invertibility

$$f \circ f^{-1} = id$$

The pointwise style readily lends itself to induction proofs while the point-free style is more suitable for purely equational reasoning. Many may also consider the pointwise

style to be more readable, despite it usually producing considerably longer code. In this thesis, we continue to use both styles freely, choosing whichever is more suitable. For completeness, we include a short introduction to the point-free style to standardize the notations we will use. For more details on this topic, please refer to [BdM97].

2.2.1 Point-free Programming

The point-free style is for programming. Nevertheless, its standard combinators have deep roots in category theory describing the algebra of datatypes. In this thesis, we do not go deeply into the algebra of programming. We look at the combinators purely from a programming language perspective, and try to utilize them as the basis of language designs.

The most fundamental combinator is functional composition $\circ$, where two functions are applied in sequence to yield a combined result. The function *id* is simply the identity.

$$
\begin{aligned}
&id :: A \to A \\
&\circ ::(B \to C) \to (A \to B) \to (A \to C)
\end{aligned}
$$

Products (implemented as pairs) are one of the primitive structures, which represent parallel existence of two pieces of data. The elements of a pair can be retrieved by projection functions *fst* and *snd*, and the function $\vartriangle$ (pronounced split) constructs a pair by applying two functions separately to a single datum.

$$
\begin{aligned}
&fst :: (A, B) \to A \\
&fst\ (x, y)\ = x \\
&snd :: (A, B) \to B \\
&snd\ (x, y) = y \\
&\vartriangle ::(A \to B) \to (A \to C) \to (A \to (B, C)) \\
&f \vartriangle g \qquad = \lambda x \to (f\ x, g\ x)
\end{aligned}
$$

Datatypes can be constructed by injecting an alternative into a sum type, represented by *Either* in Haskell; and a sum type can be consumed by $\triangledown$ (pronounced junc), which applies either of its two input functions depending on the alternative encapsulated in the sum.

$$
\begin{aligned}
&inl :: A \to Either\ A\ B \\
&inl\ x\ = Left\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inr} :: B \to \mathit{Either}\ A\ B \\
&\mathit{inr}\ x = \mathit{Right}\ x \\
&\triangledown ::(A \to C) \to (B \to C) \to (\mathit{Either}\ A\ B \to C) \\
&f \triangledown g = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \mathit{Left}\ a \quad \to f\ a \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Right}\ a \to g\ a
\end{aligned}
$$

With sums, products and recursion, we are able to express regular datatypes. For example, the standard list datatype can be defined as the following.

$$\mathit{List}\ a = \mathit{Either}\ ()\ (a, \mathit{List}\ a)$$

The unit type () represents the alternative of an empty list and the pair $(a, \mathit{List}\ a)$ represents the alternative of a non-empty list.

There are also some commonly seen derived combinators such as

$$
\begin{aligned}
&\mathit{swap} :: (A, B) \to (B, A) \\
&\times\ :: (A \to B) \to (C \to D) \to (A, B) \to (C, D)
\end{aligned}
$$

We will introduce them as we go along.

Structural recursion in point-free programming is usually described as a *fold*, which traverses a structure and replaces the constructors with applications of its body. As a result, the signature of a fold depends on the datatype that it deals with. For standard lists, we have the following.

$$
\begin{aligned}
&\mathit{foldL} :: (\mathit{Either}\ ()\ (a, b) \to b) \to ([a] \to b) \\
&\mathit{foldL}\ f = \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\
&\quad [\,] \qquad \to f\ (\mathit{Left}\ ()) \\
&\quad (x : xs) \to f\ (\mathit{Right}\ (x, \mathit{foldL}\ f\ xs))
\end{aligned}
$$

As an example, the size function can be defined as a *fold*.

$$
\begin{aligned}
&\mathit{size} = \mathit{foldL}\ (\mathit{zero} \triangledown \mathit{incr}) \\
&\quad \mathbf{where}\ \mathit{zero}\ () = 0 \\
&\qquad\quad \mathit{incr} \quad = (1+) \circ \mathit{snd}
\end{aligned}
$$

2.2.2 Bidirectional Properties

Given the different origins from a range of communities, the term "bidirectional" can mean different things. One well studied kind of bidirectional technique is program inversion. An *inverse* g of a function f is a function such that when composed with f either way produces the identity. Inverses are unique and a function with such an inverse is called *invertible*.

Being invertible in the above sense is certainly a very strong condition; only bijective functions are invertible. Instead of going for a full inverse, we can opt for a kind of partial inverse: a function g is a *left-inverse* of f if $g \circ f = id$ and a *right-inverse* of f if $f \circ g = id$. Therefore a full inverse is both a left- and right- inverse. Nevertheless, we may simply use "inverse" to refer to either left- or right- inverse if it is clear from the context.

Right-inverses exist for any surjective function, but are usually not unique. So there is the problem of finding a suitable one. On the other hand, left-inverses are unique for surjective functions, but only exist if injectivity is also present. These obvious limitations are exactly the reasons why function *put* in view-update takes an additional argument to recover injectivity, which gives rise to the acceptability law:

Acceptability $put\ (get\ s, s) = s$

Acceptability is the counterpart of left-invertibility in the view-update setting; this is made explicit in its point-free version first documented in [PC10].

Acceptability $put \circ (get \vartriangle id) = id$

The information of the original source made available to the putback function also plays a part in a deterministic right inversion. The counterpart of right-invertibility in view update is the consistency law.

Consistency $get\ (put\ (v, s)) = v$

Again, the correspondence is made explicit in the point-free version.

Consistency $get \circ put = fst$

It turns out that both the program inversion and view-update settings are useful in practise; we use bidirectional programming to mean either, and rely on the context to disambiguate. As a rule of thumb, Chapter 3 is based on the former while Chapter 4 and Chapter 5 are based on the latter.

In most bidirectional languages, it is the case that the programmer actively writes a uni-directional function and its "backwards" version is automatically generated. To be precise at times, we name the actively written function the *retrieval* function, and its domain and codomain as *source* and *view*. The "backwards" companion of a retrieval function is called a *putback* function, denoted with a superscript $<$. An important measurement on the effectiveness of a bidirectional system is how much of the range of the retrieval function is covered by the domain of the putback function. We call such a property *updatability*: better updatability implies more changes to the view can be mapped back to a source safely.

Chapter 3

Looking from the Right

Don't Fix Your Problems, Make Them Disappear.
(Vishen Lakhiani)

3.1 Introduction

3.1.1 Program Development

Suppose that you are developing a program involving some data structure. You don't yet know which operations you will need on the data structure, nor what efficiency constraints you will impose on those operations. Instead, you want to prototype the program, and conduct some initial experiments on the prototype; on the basis of the results from those experiments, you will decide whether a naive representation of the data structure suffices, or whether you need to choose a more sophisticated implementation. In the latter case, you do not want to have to conduct major surgery on your prototype in order to refactor it to use a different representation.

The traditional solution to this problem is to use data abstraction: identify (or evolve) an interface, program to that interface, and allow the implementation to vary without perturbing the program. However, that requires you to prepare in advance for the possible change of representation: it doesn't provide a smooth revision path if you didn't have the foresight to introduce the interface in the first place, but used a bare algebraic datatype as the representation.

Moreover, choosing a naive representation in terms of an algebraic datatype has considerable attractions. Programs that manipulate the data can be defined using *pattern matching* over the constructors of the datatype, rather than having to use 'observer' operations on a data abstraction. This leads to a concise and elegant programming style, which being based on equations is especially convenient for reasoning about program behaviour [Wad87b].

3.1.2 Pattern Matching

As a simple example, consider encoding binary numbers as lists of bits, most significant first:

> **data** $Bin = Zero \mid One$
> **type** $Num = [Bin]$

The above declarations introduce a new datatype (Bin) for binary bits, and define Num as a type synonym for lists of bits. Throughout this paper, we will use the syntax and standard prelude functions of Haskell [PJ03] for illustration; but any language providing algebraic datatypes would work just as well.

Consider this function for normalizing binary numbers by eliding leading zeroes, defined by pattern matching.

$$
\begin{array}{lll}
\mathit{normalize} :: \mathit{Num} \to \mathit{Num} & & \\
\mathit{normalize}\ [\,] & = [\,] & \text{-- Clause (1)} \\
\mathit{normalize}\ (\mathit{One} : \mathit{num}) & = \mathit{One} : \mathit{num} & \text{-- Clause (2)} \\
\mathit{normalize}\ (\mathit{Zero} : \mathit{num}) & = \mathit{normalize}\ \mathit{num} & \text{-- Clause (3)}
\end{array}
$$

The definition forms a collection of equations, which give a straightforward explanation of the operational behaviour of the function:

$$
\begin{array}{ll}
 & \mathit{normalize}\ [\mathit{Zero}, \mathit{One}, \mathit{Zero}] \\
= & \quad \{\, \text{Clause (3)} \,\} \\
 & \mathit{normalize}\ [\mathit{One}, \mathit{Zero}] \\
= & \quad \{\, \text{Clause (2) and Clause (1)} \,\} \\
 & [\mathit{One}, \mathit{Zero}]
\end{array}
$$

They are also convenient for calculation; for example, here is one case of an inductive proof that *normalize* is idempotent:

$$
\begin{array}{ll}
 & \mathit{normalize}\ (\mathit{normalize}\ (\mathit{Zero} : \mathit{num})) \\
= & \quad \{\, \text{Clause (3)} \,\} \\
 & \mathit{normalize}\ (\mathit{normalize}\ \mathit{num}) \\
= & \quad \{\, \text{inductive hypothesis} \,\} \\
 & \mathit{normalize}\ \mathit{num} \\
= & \quad \{\, \text{Clause (3)} \,\} \\
 & \mathit{normalize}\ (\mathit{Zero} : \mathit{num})
\end{array}
$$

An equivalent pointwise definition without using pattern matching is harder to read:

$$
\begin{array}{l}
\mathit{normalize} :: \mathit{Num} \to \mathit{Num} \\
\mathit{normalize}\ \mathit{num} = \textbf{if}\ \mathit{null}\ \mathit{num} \vee \mathit{one}\ (\mathit{head}\ \mathit{num})\ \textbf{then} \\
\qquad\qquad\qquad\quad \mathit{num} \\
\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\quad \mathit{normalize}\ (\mathit{tail}\ \mathit{num})
\end{array}
$$

It is also much less convenient for calculating with.[1]

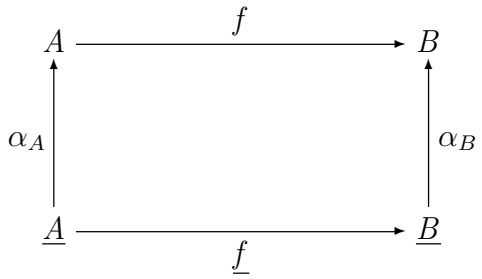
[1]Note that we don't compare with the pointfree programming style, as the discussion here is between pointwise programming with and without pattern matching.

Pattern matching has accordingly been supported as a standard feature in most modern functional languages, since its introduction in Hope [BMS80]. More recently, it has started gaining recognition from the object-oriented community too [EOW07, MRV03, LM03]. Unfortunately, the appeal of pattern matching wanes when we need to change the implementation of a data structure: function definitions are tightly coupled to a particular representation, and a change of representation has a far-reaching effect. As a result, it has been observed that the wide spread of pattern matching "*leads to a discontinuity in programming: programmers initially use pattern matching heavily, and are then forced to abandon the technique in order to regain abstraction over representations*" [SNM07].

In this stand-off between clarity and abstraction, functional languages usually lean to the former while object-oriented languages prefer the latter. Can we be a bit more ambitious, and hope for the best of both worlds? With current technology, encapsulating datatypes in a functional language excludes pattern matching with sound reasoning immediately. In an object-oriented setting, there is a concept of *typical object* [LG00], which is a simple and general data structure (often expressed in a way similar to an algebraic datatype) used as a model of the underlying more complicated and user-defined structure. For example, various kinds of linear structure can be modeled as a list datatype. The user-defined structures are used for execution while the typical object caters for specification. The function that maps an underlying structure to its typical object is called an *abstraction function*, with which an implementation using the underlying structure can be proven correct, with respect to its specification using the typical object [Mil71, Hoa72]. We look at the idea in more detail next.

3.2 Data Abstraction

Choosing the right data structure is key to achieving an efficient program; data abstraction allows us to defer the choice of representation until after the uses of the data are fully understood. The idea is to firstly program with "abstract" data, which is then replaced by a more efficient "concrete" representation. Based on the new concrete representation, operations are implemented with the original abstract program serving as specifications. It is pointed out by Milner [Mil71] that the above transition step can be proven correct by showing the following square commutes,



a condition also known as the *promotion* condition [Bir84]

$$\alpha_B \circ \underline{f} \;=\; f \circ \alpha_A$$

Here, the α_X family of functions are *abstraction* functions that map the concrete representations to the abstract ones; we will simply write α if its type is clear from the context. We use the naming convention of underlining an abstract program's name to denote its concrete implementation. In the sequel, we use the terms "abstract program" and "specification" interchangeably.

3.2.1 An Example

Consider queue structures with an abstract representation as lists.

$$\begin{aligned}
&\textbf{type } \mathit{Queue}\ a = [a] \\
&\mathit{emptyQ} = [\,] \\
&\mathit{first} \quad = \mathit{head} \\
&\mathit{isEmpty} = \mathit{null} \\
&\mathit{enQ}\ a\ q = q +\!\!+ [a] \\
&\mathit{deQ} \quad = \mathit{tail}
\end{aligned}$$

For a more efficient definition of enQ, a plausible concrete representation reads:

$$\begin{aligned}
&\textbf{type } \underline{\mathit{Queue}}\ a = ([a], [a]) \\
&\alpha :: \underline{\mathit{Queue}}\ a \to \mathit{Queue}\ a \\
&\alpha\ (fq, bq) = fq +\!\!+ \mathit{reverse}\ bq
\end{aligned}$$

The second list of the pair, representing the latter part of a queue, is reversed, so that enqueuing simply prefixes an element onto it. The library operations can be implemented as follows:

$$
\begin{aligned}
\underline{emptyQ} &= ([\,], [\,]) \\
\underline{first}\ ([\,], bq) &= last\ bq \\
\underline{first}\ ((a : fq), bq) &= a \\
\underline{isEmpty}\ ([\,], [\,]) &= True \\
\underline{isEmpty}\ q &= False \\
\underline{enQ}\ a\ (fq, bq) &= (fq, a : bq) \\
\underline{deQ}\ ([\,], bq)) &= \underline{deQ}\ (reverse\ bq, [\,]) \\
\underline{deQ}\ (a : fq, bq) &= (fq, bq)
\end{aligned}
$$

Through standard equational reasoning, we can establish the correctness of the implementation by proving the promotion condition. For example,

$$
\begin{aligned}
& \alpha\ (\underline{deQ}\ (a : fq, bq)) \\
= & \quad \{\,\text{Definition of } \underline{deQ}\,\} \\
& \alpha\ (fq, bq) \\
= & \quad \{\,\text{Definition of } deQ\,\} \\
& deQ\ (a : \alpha\ (fq, bq)) \\
= & \quad \{\,\text{Definition of } \alpha\,\} \\
& deQ\ (a : fq +\!\!+ reverse\ bq) \\
= & \quad \{\,\text{Definition of } +\!\!+\,\} \\
& deQ\ ((a : fq) +\!\!+ reverse\ bq) \\
= & \quad \{\,\text{Definition of } \alpha\,\} \\
& deQ\ (\alpha\ (a : fq, bq))
\end{aligned}
$$

Other than the handful of "primitive" operations above, we have other abstract programs benefitting from the simple list representation. For example, the map function on queues:

$$
\begin{aligned}
& mapQ :: (a \to b) \to Queue\ a \to Queue\ b \\
& mapQ\ f\ [\,] \quad = [\,] \\
& mapQ\ f\ (a : q) = f\ a : mapQ\ f\ q
\end{aligned}
$$

or a prioritisation function, which is essentially a stable sort based on element weight:

$$
\begin{aligned}
& prioritise :: Ord\ a \Rightarrow Queue\ a \to Queue\ a \\
& prioritise\ [\,] \quad = [\,] \\
& prioritise\ (a : q) = insert\ a\ (prioritise\ q)
\end{aligned}
$$

$$\begin{aligned}\textbf{where } insert\ b\ [\,] &= [b]\\ insert\ b\ (a:q) &= \textbf{if } b \leqslant a \textbf{ then } b:(a:q)\\ &\quad\ \textbf{else} \qquad\quad a:(insert\ b\ q)\end{aligned}$$

To maintain executability, all uses of the abstract representation have to be changed at once, even though some of the old definitions may not gain from the refactoring. One has to (re)implement all the functions either by pattern matching on the new representation (a time bomb for further refactoring) or by the use of primitive operations (at the cost of losing convenient reasoning). For example, a definition of map using only the primitive operations is likely to be more clumsy:

$$\begin{aligned}&mapQ_{\text{prim}} :: (a \to b) \to Queue\ a \to Queue\ b\\ &mapQ_{\text{prim}}\ f\ q = accum\ f\ q\ emptyQ\\ &\quad \textbf{where } accum\ f\ q\ aq = \textbf{if } isEmpty\ q \textbf{ then}\\ &\qquad\qquad\qquad\qquad aq\\ &\qquad\qquad\qquad\quad \textbf{else}\\ &\qquad\qquad\qquad\qquad accum\ f\ (deQ\ q)\ (enQ\ (f\ (first\ q))\ aq)\end{aligned}$$

The purpose of this chapter is to allow incremental refactoring of a program, replacing a specification by a more sophisticated implementation. We have seen previously that once an implementation is given, the promotion condition is sufficient to guarantee its correctness. However, the promotion condition does not suggest a way of integrating the new definition into its original context, which may still operate on the old representation. We need a computation law of the form

$$f = \alpha \circ \underline{f} \circ \gamma$$

in order to replace specification f with its implementation $\alpha \circ \underline{f} \circ \gamma$.

Here, the function γ does the "opposite" of α, mapping a value in the specification to one in the implementation. During execution, any value constructed in the specification is firstly converted to a value in the implementation before being passed to the concrete program; and the resulting output is converted back to the specification. For example, given the program $deQ\ [1]$, what really executes is $(\alpha \circ \underline{deQ} \circ \gamma)\ [1]$, where the two functions γ and α convert a value of type $Queue$ into one of type $\underline{Queue}$ and back again.

It is obvious that if α and γ are each other's inverses (also known as *strong simulation* in [Mil71]), the computation law is equivalent to the promotion condition. However, requiring them to be inverses restricts the abstraction function to be bijective, which

is rather a strong condition. In fact, one-sided invertibility is sufficient in this case. Specifically, the γ function should be a right inverse of α – that is, $\alpha \circ \gamma = id$.

To come out with the α/γ pair is certainly a bidirectional programming problem. In the next Section, we will sketch a proposal for a right-invertible language, and use it (Section3.4) in a framework where refactoring can be done selectively; and at any point in the process, executability and reasoning are fully supported.

3.3 The Right-Invertible Language 'RINV'

A standard way of defining a bidirectional language is to express it as a combinator library, where the bidirectional properties of the primitive functions are preserved by the set of combinators. Instead of proposing some new and stylized combinators, we base our design on the established Haskell pointfree programming framework [BdM97, Section 2.2.1], and try to identify the part of the framework that preserves right-invertibility. We work in the setting of total functions, so that $\alpha \circ \gamma = id$. The syntax of our language RINV is as below. (Non-terminals are indicated in small capitals.)

Language	RINV	::= CSTR ▌ PRIM ▌ COMB
Constructors	CSTR	::= *nil* ▌ *cons* ▌ *snoc* ▌ *wrap*
Primitives	PRIM	::= *app* ▌ *id* ▌ *assocr* ▌ *assocl* ▌ *swap* ▌ *fst*. *g* ▌ *snd*. *g*
Combinators	COMB	::= RINV ∘ RINV ▌ $fold_X$ RINV ▌ $unfold_X$ RINV ▌
		RINV ▽ RINV ▌ RINV × RINV

We purposely keep the language small as a demonstration of idea. In contrast to the point-free style of programming, RINV has the additional feature that a right inverse is automatically generated for each function that is constructed. As a result, a definition $f :: s \leftrightarrows t$ in RINV actually represents a pair of functions (hence the notation $\leftrightarrows$): the forward function $[\![f]\!] :: s \to t$, and its right inverse $[\![f]\!]^< :: t \to s$, which together satisfy $[\![f]\!] \circ [\![f]\!]^< = id$. However, for convenience when clear from context, we don't distinguish between f and its forward function $[\![f]\!]$.

The generated right inverses are intended to be total, so the forward functions have to be surjective; this property holds of the primitive functions (except for individual constructors of a multi-variant algebraic datatype—see below) and is preserved by the combinators.

There is an extensible set of primitive functions (cf. Section 3.3.1) defining the basic non-terminal building blocks of the language. Any surjective function could be made a

primitive in RINV. All primitive functions are uncurried; this fits better with the invertible framework, where a clear distinction between input and output is required. For the sake of demonstration, we present a small but representative collection of primitive functions above: *swap*, *assocl*, and *assocr* rearrange the components of an input pair; *id* is the identity operation; *app* is the uncurried append function on lists; and fst_g and snd_g are projection functions on pairs – discarding an element that can be regenerated by applying function g to the other one. As we will show, with just these few we can define many interesting functions.

The set of constructor functions (cf. Section 3.3.2) is also extensible, growing with the introduction of new datatypes. We use lowercase names for the uncurried versions of constructors. In addition to the left-biased list constructor *cons* that comes with the usual datatype declaration, we also include its right-biased counterpart *snoc*, which adds an element at the end; it can be defined in Haskell as

$$snoc = \lambda(x, xs) \to xs +\!\!+ [x]$$

Another additional constructor for lists is *wrap*, which creates a singleton list.

$$wrap\ x = [x]$$

This ability to admit functions that do not directly arise from a datatype declaration as constructors is crucial for the expressiveness of RINV. Although this might seem ad-hoc, it is by no means arbitrary. One should only use functions that truly model a different representation of the datatype. For example, *snoc* and *nil* form the familiar backwards representation of lists, while *wrap*, *nil* and the primitive function *app* correspond to the join list representation [Bir87].

The reason for keeping constructor functions separated from primitive functions is that they are non-surjective in nature. We have to relax the surjectivity rule for constructors, and require that lone constructors must be combined with other functions by the 'junc' combinator $\triangledown$, which dispatches to one of two functions according to the result of matching on a sum (cf. Section 3.3.3.1). When one of the operands of $\triangledown$ is surjective, or the two operands cover both constructors of a two-variant datatype, the result is surjective; such pairs of operands are *jointly surjective* [MB04]. If we see types as sets of values, two functions $f :: a \to c$ and $g :: b \to c$ being jointly surjective is defined as $range\ (f) \cup range\ (g) = c$. For example, $nil \triangledown cons$ and $nil \triangledown id$ are both surjective, but $cons \triangledown snoc$ is not. Since $\triangledown$ can be nested, this result extends to datatypes with more than two constructors. Constructor functions can be composed with other functions as

well, using the standard function composition combinator $\circ$, but only to the left (cf. Section 3.3.3.1): once a non-surjective function appears in a chain of compositions other than in the leftmost position, it is difficult to analyse the exact range of the composition. Both the above requirements can be enforced by a rather straightforward syntactic check, which we will briefly sketch when discussing the individual combinators later.

Other than the two already mentioned combinators, $\times$ is the cartesian product of two functions (cf. Section 3.3.3.1), and $fold_X\ f$ is the operator for regular structural recursion (with $unfold_X$ as its dual), which decomposes a structure of type X and replaces the constructors with its body f (cf. Section 3.3.3.2); very often we omit the subscript X, when it is understood from context or irrelevant. In combination with *swap*, *assocl* and *assocr*, $\times$ is able to define all functions that rearrange the components of a tuple, while $\triangledown$ is useful in constructing the body of a *fold*. We don't include $\triangle$, the dual of $\triangledown$, in RINV, because of surjectivity, as will be explained shortly.

With the language RINV, we can state the following property.

Theorem 3.1 (Right invertibility) *Given a function f in* RINV*, $[\![f]\!] \circ [\![f]\!]^< = id$.*

The correctness of this theorem should become evident by the end of this section, as we discuss in detail the various constructs of RINV and their properties.

3.3.1 The Primitive Functions

The function *id* is the identity function; functions *assocr*, *assocl* and *swap* manipulate pairs.

$$
\begin{aligned}
&assocr :: ((a, b), c) \leftrightarrows (a, (b, c)) \\
&[\![assocr]\!] \ = \lambda((a, b), c) \rightarrow (a, (b, c)) \\
&[\![assocr]\!]^< = [\![assocl]\!] \\
&assocl :: (a, (b, c)) \leftrightarrows ((a, b), c) \\
&[\![assocl]\!] \ = \lambda(a, (b, c)) \rightarrow ((a, b), c) \\
&[\![assocl]\!]^< = [\![assocr]\!] \\
&swap :: (a, b) \leftrightarrows (b, a) \\
&[\![swap]\!] = \lambda(a, b) \rightarrow (b, a) \\
&[\![swap]\!]^< = [\![swap]\!]
\end{aligned}
$$

Together with the combinators $\times$ and $\circ$, these are sufficient to define many functions on pairs. For example,

$$
\begin{aligned}
&subr &&:: (b, (a, c)) \leftrightarrows (a, (b, c)) \\
&subr &&= assocr \circ (swap \times id) \circ assocl \\
&trans &&:: ((a, b_1), (b_2, c)) \leftrightarrows ((a, b_2), (b_1, c)) \\
&trans &&= assocl \circ (id \times subr) \circ assocr
\end{aligned}
$$

Function *app* is the uncurried append function, which is not injective. The admission of non-injective functions is one of the most important distinctions between RINV and other invertible languages [MHT04a], allowing us to break away from the isomorphism restriction. There are many possible right inverses for *app*, of which we pick one:

$$[\![app]\!]^{<} = \lambda xs \rightarrow splitAt\ ((length\ xs + 1)\ `div`\ 2)\ xs$$

For data projection functions such as *fst* and *snd*, a legitimate way of inverting them is by filling the discarded element with an arbitrary value. Though satisfying right invertibility, pairs created this way have little practical value for obvious reasons. Therefore, we restrict the use of projection functions to the cases where the discarded values are recoverable, which gives rise to the following.

$$
\begin{aligned}
&fst_g :: (a \rightarrow b) \rightarrow (a, b) \leftrightarrows a \\
&[\![fst_g]\!]^{<} = \lambda x \rightarrow (x, g\ x)
\end{aligned}
$$

and

$$
\begin{aligned}
&snd_g :: (b \rightarrow a) \rightarrow (a, b) \leftrightarrows b \\
&[\![snd_g]\!]^{<} = \lambda y \rightarrow (g\ y, y)
\end{aligned}
$$

It is made clear that fst_g and snd_g remove duplications from a pair and reintroduce them in the other direction. Note that the function g itself is not necessarily in RINV; it just needs to be definable in the host language (in our case, Haskell).

3.3.2 The Constructors

The semantics of the constructor functions are simple: they follow directly from the corresponding constructors introduced by datatype declarations, except for being uncurried. For example,

$$
\begin{aligned}
&[\![nil]\!] &&= \lambda() \rightarrow [\,] \\
&[\![cons]\!] &&= \lambda(x, xs) \rightarrow x : xs
\end{aligned}
$$

Constructors *snoc* and *wrap* are not primitive on left-biased lists, but can be encoded:

$$[\![snoc]\!] = \lambda(xs, x) \to xs +\!\!+ [x]$$
$$[\![wrap]\!] = \lambda x \to [x]$$

Inverses of the primitive constructor functions are obtained simply by swapping the right- and left-hand sides of the definitions. For example, we have

$$[\![nil]\!]^< \;\; = \lambda[\,] \qquad \to ()$$
$$[\![cons]\!]^< = \lambda(x : xs) \to (x, xs)$$

The right inverses of *snoc* and *wrap* are

$$[\![snoc]\!]^< [x] \qquad = ([\,], x)$$
$$[\![snoc]\!]^< (x : xs) = \textbf{let } (ys, y) = [\![snoc]\!]^< xs \textbf{ in } (x : ys, y)$$
$$[\![wrap]\!]^< [x] \qquad = x$$

3.3.3 The Combinators

The combinators in RINV are mostly standard.

3.3.3.1 Composition, Sum and Product

Combinator $\circ$ sequentially composes two functions:

$$[\![f \circ g]\!] \;\; = [\![f]\!] \circ [\![g]\!]$$
$$[\![f \circ g]\!]^< = [\![g]\!]^< \circ [\![f]\!]^<$$

Its inverse is the reverse composition of the inverses of the two arguments.

Combinators $\times$ and $\triangledown$ compose functions in parallel. The former applies a pair of functions component-wise to its input:

$$(\times) :: (a \leftrightarrows b) \to (c \leftrightarrows d) \to ((a, c) \leftrightarrows (b, d))$$
$$[\![f \times g]\!] \;\; = \lambda(w, x) \to ([\![f]\!]\ w, [\![g]\!]\ x)$$
$$[\![f \times g]\!]^< = \lambda(y, z) \to ([\![f]\!]^<\ y, [\![g]\!]^<\ z)$$

Note that we have chosen not to define $\times$ in terms of a more primitive combinator $\triangle$, which executes both of its input functions on a single datum:

$$(\triangle) :: (a \leftrightarrows b) \to (a \leftrightarrows c) \to (a \leftrightarrows (b, c))$$
$$[\![f \triangle g]\!] = \lambda x \to ([\![f]\!]\ x, [\![g]\!]\ x)$$

However, in the inverse direction, $[\![f]\!]^<\ x$ and $[\![g]\!]^<\ y$ would have to converge, which is difficult to enforce statically. Indeed, functions constructed with $\triangle$ are generally not surjective, and so do not have total right inverses; for this reason, we exclude $\triangle$ from RINV.

The combinator $\triangledown$ consumes an element of a sum type ('*Either*' in Haskell).

$$(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (\mathit{Either}\ a\ b \leftrightarrows c)$$
$$[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Left}\ a \to [\![f]\!]\ a\ ;\ \mathit{Right}\ b \to [\![g]\!]\ b\}$$

In the inverse direction, if both f and g are surjective, it doesn't matter which branch is chosen. However, the use of constructor functions deserves some attention, since they are not surjective in isolation. In contrast to the case of $\triangle$, here the totality in the inverse direction can be recovered by choosing a non-failing branch. As a result, in the event that $[\![f]\!]^<$ fails on certain inputs, $[\![g]\!]^<$ should be applied. To model this failure handling, we lift functions in RINV into the *Maybe* monad (allowing an extra possibility for the return value), and handle a failure in the first function by invoking the second.

$$[\![f \triangledown g]\!]^< = \lambda x \to ([\![f]\!]^<\ x)\ \text{`}\mathit{choice}\text{`}\ ([\![g]\!]^<\ x)$$
$$\mathit{choice} :: \mathit{Maybe}\ a \to \mathit{Maybe}\ a \to \mathit{Maybe}\ a$$
$$\mathit{choice}\ (\mathit{Just}\ x)\ _ = x$$
$$\mathit{choice}\ _\ (\mathit{Just}\ y) = y$$

This shallow backtracking is sufficient because the guards of conditionals are only pattern matching outcomes, which are completely decided at each level. Note that for the sake of presentation, in the sequel of the chapter, we still use the non-monadic types for $f \triangledown g$, with the understanding that all functions in RINV are lifted to the *Maybe* monad in an implementation.

In general, it is not an easy task to check surjectivity of functions. However, in RINV, this test is made relatively straightforward, since the only possible cause for $f \triangledown g$ not to be surjective is that both f and g use constructor functions; in this case, it is clear that we need the complete set of constructors to satisfy the condition of surjectivity.

The more intricate part is to analyse the surjectivity of the composition (and hence the totality of its inverse). It is clear that if one of the functions in a chain of compositions is not surjective, the composed function may also be non-surjective. However, there is no

easy way of determining the range of such a composition if the non-surjective function is not the leftmost one in the chain, which makes it unsuitable for constructing jointly surjective functions through the $\triangledown$ combinator as discussed above. Therefore, in RINV, we disallow compositions involving constructor functions on the right of a composition.

3.3.3.2 Recursion

With the ground prepared, we are now ready to discuss recursive combinators. We define

$$[\![fold_X\ f]\!]^< = unfold_X\ [\![f]\!]^<$$
$$[\![unfold_X\ f]\!]^< = fold_X\ [\![f]\!]^<$$

The forward semantics of $fold_X\ f$ is the standard *fold* for a datatype X; its inverse semantics is defined by a corresponding $unfold_X$, and vice versa. In this paper, we may overload *fold* and *unfold* when the datatype is understood or irrelevant. Intuitively, *fold* disassembles a structure and replaces the constructors with applications of the body. Function *unfold*, on the other hand, takes a seed, splitting it with the body into building blocks of a structure and new seeds, which are themselves recursively unfolded. In short, *fold* collapses a structure, whereas *unfold* grows one.

When an algebraic datatype X is given, Haskell definitions of $fold_X$ and $unfold_X$ can be generated. For example, consider the datatype of lists:

$$\begin{aligned}
&fold_{List} :: (Either\ ()\ (a, b) \to b) \to (List\ a \to b)\\
&fold_{List}\ f = \lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\qquad [\,] \qquad\ \to f\ (Left\ ())\\
&\qquad\qquad (x : xs) \to f\ (Right\ (x, (fold_{List}\ f\ xs)))\\
&unfold_{List} :: (b \to Either\ ()\ (a, b)) \to (b \to List\ a)\\
&unfold_{List}\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of}\ Left\ () \qquad\ \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad Right\ (a, b) \to a : (unfold_{List}\ f\ b)
\end{aligned}$$

Another example is leaf-labeled binary trees. Note that the constructor *Fork* is uncurried to fit better into the RINV framework.

$$\begin{aligned}
&\mathbf{data}\ LTree\ a = Leaf\ a \mid Fork\ (LTree\ a, LTree\ a)\\
&fold_{LTree} :: (Either\ a\ (b, b) \to b) \to LTree\ a \to b\\
&fold_{LTree}\ f = \lambda t \to \mathbf{case}\ t\ \mathbf{of}\\
&\quad Leaf\ a \qquad\ \to f\ (Left\ a)\\
&\quad Fork\ (t_1, t_2) \to f\ (Right\ (fold_{LTree}\ f\ t_1, fold_{LTree}\ f\ t_2))
\end{aligned}$$

$$
\begin{aligned}
&\mathit{unfold}_{\mathit{LTree}} :: (a \to \mathit{Either}\ a\ (b, b)) \to b \to \mathit{LTree}\ a \\
&\mathit{unfold}_{\mathit{LTree}}\ f = \lambda b \to \mathbf{case}\ f\ b\ \mathbf{of} \\
&\qquad\qquad \mathit{Left}\ a \qquad\quad \to \mathit{Leaf}\ a \\
&\qquad\qquad \mathit{Right}\ (b_1, b_2) \to \mathit{Fork}\ (\mathit{unfold}_{\mathit{LTree}}\ f\ b_1, \mathit{unfold}_{\mathit{LTree}}\ f\ b_2)
\end{aligned}
$$

We use *fold* and *unfold* to construct the right inverses of each other. A subtlety here is that in the setting of Set and total functions, which this thesis is based on, the initial algebra does not coincide with the final coalgebra, resulting in incompatible folds and unfolds. A standard fix to this problem is to restrict the coalgebras to *recursive* ones [CUV06], which allows folds and unfolds to compose [HHJ11]. With recursive coalgebras, we have the following results [PC10]:

Fact 3.2 $\mathit{unfold}\ [\![f]\!] \circ \mathit{fold}\ [\![f]\!]^{<} = \mathit{id}.$

Fact 3.3 $\mathit{fold}\ [\![f]\!] \circ \mathit{unfold}\ [\![f]\!]^{<} = \mathit{id}.$

3.3.4 Programming in RINV

We are now ready to look into a few examples of the kinds of function we can define with RINV.

To start with, let's look first at a very useful derived combinator *map* that can be defined in term of *fold*. For example, *map* on lists, $\mathit{map}_{\mathit{List}}$, is defined as follows.

$$
\begin{aligned}
&\mathit{map}_{\mathit{List}} :: (a \leftrightarrows b) \to (\mathit{List}\ a \leftrightarrows \mathit{List}\ b) \\
&\mathit{map}_{\mathit{List}}\ f = \mathit{fold}_{\mathit{List}}\ (\mathit{nil} \triangledown (\mathit{cons} \circ (f \times \mathit{id})))
\end{aligned}
$$

Function $\mathit{map}_{\mathit{List}}\ f$ applies argument f uniformly to all the elements of a list, without modifying the list structure. Since *nil* and *cons* form a complete set of constructors for lists, we know they are jointly surjective.

Similarly, *map* on leaf-labeled trees, $\mathit{map}_{\mathit{LTree}}$, is defined as follows.

$$
\begin{aligned}
&\mathit{map}_{\mathit{LTree}} :: (a \leftrightarrows b) \to (\mathit{Tree}\ a \leftrightarrows \mathit{Tree}\ b) \\
&\mathit{map}_{\mathit{LTree}}\ f = \mathit{fold}_{\mathit{LTree}}\ ((\mathit{leaf} \circ f) \triangledown \mathit{fork})
\end{aligned}
$$

The function *reverse* on lists can be defined as a fold:

$$
\begin{aligned}
&\mathit{reverse} \quad\ = \mathit{fold}_{\mathit{List}}\ (\mathit{nil} \triangledown \mathit{snoc}) \\
&[\![\mathit{reverse}]\!]^{<} = \mathit{unfold}_{\mathit{List}}\ [\![\mathit{nil} \triangledown \mathit{snoc}]\!]^{<}
\end{aligned}
$$

In the forward direction, a list is taken apart and the first element is appended to the rear of the output list by *snoc*. This process terminates on reaching an empty list, when an empty list is returned as the result. Function $[\![snoc]\!]^<$ extracts the last element in a list and adds it to the front of the result list by *unfold*, which terminates when $[\![nil]\!]^<$ can be successfully applied (i.e when the input is the empty list). Since *nil* and *snoc* form a complete set of constructors for lists, they are jointly surjective.

Function *reverse* is also used to construct the *apprev* function that reverses a list and appends it.

$$\begin{aligned} &apprev :: ([a], [a]) \to [a] \\ &apprev = app \circ (id \times reverse) \end{aligned}$$

Function *apprev* reverses the second list before concatenating the two. For example, we have:

$$apprev\ ([1,2],[3,4,5,6,7]) = [1,2,7,6,5,4,3]$$

The companion $apprev^\circ$ function is

$$\begin{aligned} &[\![apprev]\!]^< :: [a] \to ([a], [a]) \\ &[\![apprev]\!]^< = [\![app \circ (id \times reverse)]\!]^< \end{aligned}$$

In the inverse direction, a list is split into two, and functions $[\![id]\!]^<$ and $[\![reverse]\!]^<$ are applied to the two parts. For example, we have

$$\begin{aligned} apprev\ ([\![apprev]\!]^<\ ([1,2,7,6,5,4,3])) &= apprev\ ([1,2,7,6],[3,4,5]) \\ &= [1,2,7,6,5,4,3] \end{aligned}$$

On the other hand,

$$\begin{aligned} [\![apprev]\!]^<\ (apprev\ ([1,2],[3,4,5,6,7])) &= [\![apprev]\!]^<\ ([1,2,7,6,5,4,3]) \\ &= ([1,2,7,6],[3,4,5]) \end{aligned}$$

It is clear from above that $[\![apprev]\!]^<$ is not a left inverse of *apprev*, and it is not intended to be.

Our last example is the traversal of node-labelled binary trees.

$$\textbf{data}\ BinTree\ a = BLeaf \mid BNode\ a\ (BinTree\ a, BinTree\ a)$$

The fold and unfold functions for binary trees are as follows.

$$
\begin{aligned}
&\mathit{fold}_B :: (\mathit{Either}\ ()\ (a,(b,b)) \to b) \to (\mathit{BinTree}\ b \to b)\\
&\mathit{fold}_B\ f = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad \mathit{BLeaf}\qquad\quad \to f\ (\mathit{Left}\ ())\\
&\qquad\qquad \mathit{BNode}\ a\ (l,r) \to f\ (\mathit{Right}\ (a,(\mathit{fold}_B\ f\ l, \mathit{fold}_B\ f\ r)))\\
&\mathit{unfold}_B :: (b \to \mathit{Either}\ ()\ (a,(b,b))) \to (b \to \mathit{BinTree}\ b)\\
&\mathit{unfold}_B\ f =\\
&\quad \lambda x \to \mathbf{case}\ f\ x\ \mathbf{of}\\
&\qquad \mathit{Left}\ ()\qquad\quad \to \mathit{BLeaf}\\
&\qquad \mathit{Right}\ (a,(l,r)) \to \mathit{BNode}\ a\ (\mathit{unfold}_B\ f\ l, \mathit{unfold}_B\ f\ r)
\end{aligned}
$$

Using the fold_B combinator, pre- and post-order traversal of a binary tree can be defined as follows.

$$
\begin{aligned}
\mathit{preOrd}\ &= \mathit{fold}_B\ (\mathit{nil} \triangledown (\mathit{cons} \circ (\mathit{id} \times \mathit{app})))\\
\mathit{postOrd} &= \mathit{fold}_B\ (\mathit{nil} \triangledown (\mathit{snoc} \circ \mathit{swap} \circ (\mathit{id} \times \mathit{app})))
\end{aligned}
$$

In the forward direction, fold_B adds the node value at one end of the concatenation of the two subtrees' traversals. In the inverse direction, a node value is extracted from the input list, and the rest of the list is divided and grown into individual trees. Since our choice of $[\![\mathit{app}]\!]^{<}$ splits a list in the middle, the trees constructed by the above functions will be balanced.

As a remark, the primitive function *app* can be defined in Haskell with $\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$ as:

$$\mathit{app} = \mathit{uncurry}\ (\mathit{flip}\ (\mathit{foldr}\ (:)))$$

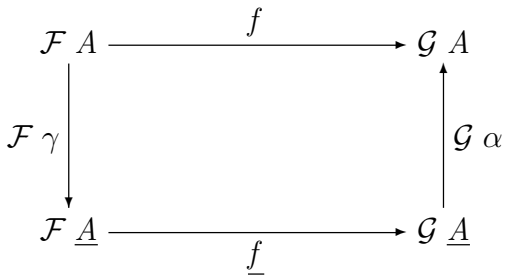
which effectively partially applies *foldr* and awaits an input as the base case. This idiom of taking an extra argument to form the base case is difficult to realize when the fold body is constructed independently, as it is in RINV.

3.4 Selective Refactoring

We are now ready to employ RINV in our goal of performing selective refactoring of functions defined with pattern matching. The queue example introduced in Section 3.2.1, will be used as the running example.

3.4.1 The Computation Law

Before going into the details of our results, we firstly generalise the notation used. The operations f in the promotion condition will not always have types as simple as $A \to A$, like deQ does. Suppose we have datatypes A and $\underline{A}$, and conversion functions $\alpha :: \underline{A} \to A$ and $\gamma :: A \to \underline{A}$. In the general case, an abstract operation will take not just a single value in the abstract representation (of type A), but some combination of abstract values and other arguments. We capture this in terms of an operation $\mathcal{F}$ on datatypes A. Similarly, the operation will return a different combination $\mathcal{G}$ of abstract values and other results. The operations $\mathcal{F}$ and $\mathcal{G}$ are functors, they lift the conversion functions in the obvious way to $\mathcal{F}\ \gamma :: \mathcal{F}\ A \to \mathcal{F}\ \underline{A}$ and $\mathcal{G}\ \alpha :: \mathcal{G}\ \underline{A} \to \mathcal{G}\ A$, respecting identity and composition. Then an abstract operation f will have type $\mathcal{F}\ A \to \mathcal{G}\ A$, and the corresponding concrete operation $\underline{f} :: \mathcal{F}\ \underline{A} \to \mathcal{G}\ \underline{A}$ should satisfy the computation law $f\ =\ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \gamma$, as shown in the following commuting diagram.



The above notation generalises to polymorphic datatypes in the standard way, by seeing $\mathcal{F}, \mathcal{G}$ as functors on the functor category (what Martin *et al.* [MGB04] call 'higher-order functors', or 'hofunctors' for short).

For example, for the operation $first :: Queue\ a \to a$, the input context $\mathcal{F}$ is the identity hofunctor, matching the source type $Queue\ a$, and the output context $\mathcal{G}$ is the constantly-identity hofunctor ($\mathcal{G}\ F = Id$), matching the target type a. The operation must satisfy the following promotion condition:

$$\underline{first}\ =\ first \circ \alpha$$

The promotion equations for the rest of the operations are listed below.

$$\begin{aligned}
\underline{isEmpty} &= isEmpty \circ \alpha \\
\alpha\ \underline{emptyQ} &= emptyQ \\
\alpha \circ \underline{enQ}\ a &= enQ\ a \circ \alpha \\
\alpha \circ \underline{deQ} &= deQ \circ \alpha
\end{aligned}$$

Given the right-invertibility property of α and γ, we can derive the computation law

$$f \;=\; \mathcal{G}\;\alpha \circ \underline{f} \circ \mathcal{F}\;\gamma$$

from the promotion condition

$$\mathcal{G}\;\alpha \circ \underline{f} \;=\; f \circ \mathcal{F}\;\alpha$$

which does not involve the function γ, as the following lemma demonstrates.

Lemma 3.4 (Computation) *Given the promotion condition* $\mathcal{G}\;\alpha \circ \underline{f} \;=\; f \circ \mathcal{F}\;\alpha$*, we can deduce the computation law* $f \;=\; \mathcal{G}\;\alpha \circ \underline{f} \circ \mathcal{F}\;\gamma$*.*

Proof.

$$\begin{array}{ll} & \mathcal{G}\;\alpha \circ \underline{f} \circ \mathcal{F}\;\gamma \\ = & \quad \{\, \mathcal{G}\;\alpha \circ \underline{f} \;=\; f \circ \mathcal{F}\;\alpha \,\} \\ & f \circ \mathcal{F}\;\alpha \circ \mathcal{F}\;\gamma \\ = & \quad \{\, \mathcal{F} \text{ respects composition} \,\} \\ & f \circ \mathcal{F}\;(\alpha \circ \gamma) \\ = & \quad \{\, \alpha \circ \gamma \;=\; id \,\} \\ & f \circ \mathcal{F}\;id \\ = & \quad \{\, \mathcal{F} \text{ respects identity} \,\} \\ & f \end{array}$$

□

The computation law requires additional infrastructure (the right inverse γ) on top of the data abstraction framework based on the promotion condition. In our design, the γs will be provided by RINV. That is to say, the programmer carrying out a refactoring writes only α, in RINV, and the corresponding γ is automatically generated. In the case of the queue example presented above, a possible definition in RINV reads:

$$\alpha = app \circ (id \times reverse)$$

3.4.2 Refactoring by Translation

In our proposal, a programmer wishing to migrate to the concrete representation has the choice of keeping the original abstract definitions, or of refactoring them into the style using only the primitive operations, or of having a mixture of the two. For example, consider a circular queue that is read for a certain amount of time, say repeatedly playing a piece of music.

$$
\begin{aligned}
&\mathit{play} :: \mathit{Time} \to \mathit{Queue}\ (\mathit{IO}\ ()) \to \mathit{IO}\ ()\\
&\mathit{play}\ 0 \qquad (a{:}\ _) \ = a\\
&\mathit{play}\ (n+1)\ (a : q) = \mathbf{do}\ a\\
&\qquad\qquad\qquad\qquad \mathit{play}\ n\ (q +\!\!+ [a])
\end{aligned}
$$

Refactoring it using the primitive operations allows us to take advantage of $\underline{enQ}$'s constant time performance.

$$
\begin{aligned}
&\mathit{play}_{\text{prim}} :: \mathit{Time} \to \underline{\mathit{Queue}}\ (\mathit{IO}\ ()) \to \mathit{IO}\ ()\\
&\mathit{play}_{\text{prim}}\ 0\ q \qquad = \underline{\mathit{first}}\ q\\
&\mathit{play}_{\text{prim}}\ (n+1)\ q = \mathbf{do}\ \mathit{hd}\\
&\qquad\qquad\qquad\qquad \mathit{play}_{\text{prim}}\ n\ (\underline{\mathit{enQ}}\ \mathit{hd}\ \mathit{tl})\\
&\quad \mathbf{where}\ \mathit{hd} = \underline{\mathit{first}}\ q\\
&\qquad\quad\ \ \mathit{tl} \ = \underline{\mathit{deQ}}\ q
\end{aligned}
$$

The standard data abstraction framework is able to verify the above implementation as correct by proving the promotion condition. Note that $\mathit{play}_{\text{prim}}$ has a slightly different type from that of *play* as it operates on the concrete representation. To perform selective refactoring, and have $\mathit{play}_{\text{prim}}$ to do the job of *play* in the original abstract context, we need to bridge this gap. For example, we may want to use $\mathit{play}_{\text{prim}}$ together with *mapQ* (defined in Section 5.1.1); or mix pattern matching with primitive operations:

$$
\begin{aligned}
&\mathit{play}_{\text{mix}}\ 0 \qquad (a : _) = a\\
&\mathit{play}_{\text{mix}}\ (n+1)\ (a : q) = \mathbf{do}\ a\\
&\qquad\qquad\qquad\qquad \mathit{play}_{\text{mix}}\ n\ (\mathit{enQ}\ a\ q)
\end{aligned}
$$

In these cases, the semantics of refactored functions can be elaborated by a mechanical translation, following a rather straightforward scheme: each use of a primitive function is replaced with its implementation precomposed with α and postcomposed with γ (subject to the appropriate (ho)functors).

The primitive operations that are defined on the concrete implementation require their inputs to be converted from the abstract representation before consumption, and the outputs converted back to the abstract representation. Effectively, all the refactored functions have the abstract representation as input and output types; the concrete representation remains only for intermediate structures. As an example, $play_{\text{mix}}$ is translated into the following.

$$\begin{aligned} play'_{\text{mix}}\ 0 \qquad (a : q) &= a \\ play'_{\text{mix}}\ (n+1)\ (a : q) &= \mathbf{do}\ a \\ &\qquad play'_{\text{mix}}\ n\ ((\alpha \circ \underline{enQ}\ a \circ \gamma)\ q) \end{aligned}$$

Given the computation law, it is easy to conclude that $play'_{\text{mix}}$ is equivalent to $play_{\text{mix}}$, in the sense that exactly the same output is produced for each input.

Theorem 3.5 *The translation is semantics-preserving.*

Proof. Follows directly from Lemma 3.4. □

The original abstract program such as the definition of *play* is turned into a specification and can be used for equational reasoning. For example, one can continue to reason

$$mapQ\ f \circ play\ n\ =\ play\ n \circ mapQ\ f$$

on the abstract level, without worrying about the refactoring of *play*.

3.4.3 Optimization

Up to now, we have achieved incremental refactoring with little additional burden for the programmer. As a result, program construction can benefit from pattern matching and straightforward proofs of correctness. The run-time performance of non-primitive functions making use only of pattern matching can be understood by standard reasoning; however, when primitive operations are called, additional conversion overhead will occur. This performance loss is to be expected for definitions such as $play_{\text{mix}}$, where an obvious switch from pattern matching to primitive operations is inevitable. However, it may be surprising that $play_{\text{prim}}$, which only involves primitive operations, is not faster. The translated code is the following.

$$\begin{aligned} play'_{\text{prim}}\ 0\ q &= (\alpha \circ \underline{first} \circ \gamma)\ q \\ play'_{\text{prim}}\ (n+1)\ q &= \mathbf{do}\ hd \end{aligned}$$

$$
\begin{aligned}
& play'_{\text{prim}}\ n\ ((\alpha \circ \underline{enQ}\ hd \circ \gamma)\ tl) \\
\textbf{where}\ hd &= (\alpha \circ \underline{first} \circ \gamma)\ q \\
tl &= (\alpha \circ \underline{deQ} \circ \gamma)\ q
\end{aligned}
$$

There are conversions everywhere in the program. It will be disastrous if all of them have to be executed. Since there is no pattern matching involved, we can try to remove the conversions through fusion. Indeed, the correctness of such fusion follows from the promotion condition. Let's take an expression fragment from the above definition for demonstration. Consider

$$(\alpha \circ \underline{enQ}\ hd \circ \gamma)\ ((\alpha \circ \underline{deQ} \circ \gamma)\ q)$$

Our target is to fuse the intermediate conversions to produce

$$(\alpha \circ \underline{enQ}\ hd \circ \underline{deQ} \circ \gamma)\ q$$

This would clearly follow from $\gamma \circ \alpha = id$, but this is not a property that we guarantee—for good reason, since requiring it in addition to the existing right inverse property $\alpha \circ \gamma = id$ demands isomorphic abstract and concrete representation, which is too restrictive to be practically useful. Instead, using the promotion condition, we can prove a weaker property that is sufficient for fusion.

Theorem 3.6 (Fusion Soundness) $\mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \gamma \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \gamma = \mathcal{H}\ \alpha \circ \underline{g} \circ \underline{f} \circ \mathcal{F}\ \gamma$.

Proof.

$$
\begin{aligned}
& \mathcal{H}\ \alpha \circ \underline{g} \circ \mathcal{G}\ \gamma \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \gamma \\
= & \quad \{\text{promotion: } \mathcal{H}\ \alpha \circ \underline{g} = g \circ \mathcal{G}\ \alpha\} \\
& g \circ \mathcal{G}\ \alpha \circ \mathcal{G}\ \gamma \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \gamma \\
= & \quad \{\mathcal{G} \text{ is a (ho)functor; } \alpha \circ \gamma = id\} \\
& g \circ \mathcal{G}\ \alpha \circ \underline{f} \circ \mathcal{F}\ \gamma \\
= & \quad \{\text{promotion: } \mathcal{H}\ \alpha \circ \underline{g} = g \circ \mathcal{G}\ \alpha\} \\
& \mathcal{H}\ \alpha \circ \underline{g} \circ \underline{f} \circ \mathcal{F}\ \gamma
\end{aligned}
$$

□

Basically, this theorem states that although the input to $\underline{g}$ may differ from the output of $\underline{f}$, due to the $\gamma \circ \alpha$ conversions, nevertheless the post-conversion of $\underline{g}$'s output brings possibly different results into the same value in the abstract representation.

In theory the above fusion result can be applied across steps of recursion, where the pipelining of conversion is not made explicit in the syntax. However, in practice for functions defined with explicit recursions, as the ones found in this section, it is rather difficult for a compiler to detect all the fusion opportunities. We leave it as future work to experiment with GHC to see how much speed up can be gained.

3.4.4 More Examples

We now look at a few additional examples making use of the refactoring framework developed in this section.

3.4.4.1 Join Lists

As an alternative to the biased linear list structure, the *join* representation of lists has been proposed for program elegance [Mee86, Bir87], efficiency [SH82], and more recently, parallelism [Ste09]. It can be defined as:

$$\textbf{data}\ \underline{List}\ a = Empty \mid Unit\ a \mid Join\ (\underline{List}\ a)\ (\underline{List}\ a)$$

As a simple example, a constant-time append function (in contrast to the linear-time left-biased-list counterpart) can be defined with this representation.

$$\underline{append}\ l1\ l2 = Join\ l1\ l2$$

At the same time, we don't want to give up on the familiar notions of [ ] and (:). Instead, they can serve as a specification of the join representation.

$$\begin{aligned}&\textbf{type}\ List\ a = [a]\\ &append\ [\,]\quad\ \ y = ys\\ &append\ (x : xs)\ ys = x : append\ xs\ ys\end{aligned}$$

A RINV definition of α is:

$$\alpha = fold\ ((nil \triangledown wrap) \triangledown app)$$

Now suppose a programmer would like to refactor certain list functions to make use of the constant-time append function; she will need to define an abstraction function $\alpha :: \underline{List}\ a \to List\ a$, and verify its correctness by proving the promotion condition. The translation outlined in Section 3.4.2 will complete the refactoring, by redirecting calls

from *append* to $\underline{append}$. At the same time, many other list functions, including some that are yet to be defined, may still use pattern matching on, and be manipulated using, the original *List* representation. For example, retrieving the head of a list can be defined as:

$$head\ (x : xs) = x$$

and calculated in the following manner, oblivious to the refactoring:

$$head\ (append\ (x : xs)\ ys)\ =\ head\ (x : append\ xs\ ys)\ =\ x$$

3.4.4.2 Binary Numbers

In the introduction, we showed a representation of binary numbers as lists of digits with the most significant bit first (MSB). This representation is intuitive, and offers good support for most operations; however, for incrementing a number, having the least significant bit (LSB) first is better.

$$\begin{aligned}
&\textbf{type}\ \underline{Num} = [Bin] \\
&\underline{incr} :: \underline{Num} \to \underline{Num} \\
&\underline{incr}\ [\,] &&= [One] \\
&\underline{incr}\ (Zero : num) &&= One : num \\
&\underline{incr}\ (One : num) &&= Zero : (\underline{incr}\ num)
\end{aligned}$$

Effectively, in order to use the above definition with any other operations, we only need to reverse the MSB representation, and a type synonym for $\underline{Num}$ can be used as documentation of this intention. However, the use of synonyms is potentially risky, as any incorrect usage won't be picked up by a compiler, because *Num* and $\underline{Num}$ are simply two different names for the same type. At the same time, defining the two representations as completely different types is very cumbersome. With our proposal, we program with only one representation (*Num*) and selectively refactor certain operations (such as *incr*), which effectively eliminates any possibility of misuse.

$$incr = \alpha \circ \underline{incr} \circ \gamma$$

The α and γ are both the reverse function on lists. As a result, only one representation is exposed to the programmer, and all the conversions between representations are handled implicitly by the refactoring translation.

A RINV definition that reverses a list reads:

$$\alpha = fold\ (nil \triangledown snoc)$$

3.4.4.3 Sized Trees

A sized annotated binary tree is suitable for fast indexing, as we can traverse it quickly by not entering any left subtree that has a smaller size than the index.

$$
\begin{aligned}
&\textbf{data}\ \mathit{STree}\ a = \mathit{Empty} \\
&\qquad\qquad \mid \mathit{SLeaf}\ a \\
&\qquad\qquad \mid \mathit{SFork}\ (\mathit{Int}, (\mathit{STree}\ a, \mathit{STree}\ a)) \\
&\underline{\mathit{index}} :: \mathit{Int} \to \mathit{STree}\ a \to a \\
&\underline{\mathit{index}}\ 1\ (\mathit{SLeaf}\ a) \qquad\qquad\qquad\qquad\quad = a \\
&\underline{\mathit{index}}\ n\ (\mathit{SFork}\ (s, (lt, rt))) \mid n > ls \quad = \underline{\mathit{index}}\ (n - ls)\ rt \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \underline{\mathit{index}}\ n\ lt \\
&\quad \textbf{where}\ ls = \mathit{getSize}\ lt \\
&\qquad\quad\ \ rs = \mathit{getSize}\ rt \\
&\mathit{getSize}\ \mathit{Empty} \qquad = 0 \\
&\mathit{getSize}\ (\mathit{SLeaf}\ _) \qquad = 1 \\
&\mathit{getSize}\ (\mathit{SFork}\ (s, _)) = s
\end{aligned}
$$

This feature of fast indexing makes sized trees an attractive alternative to lists, when access to elements in the middle is required, with the following specification.

$$
\begin{aligned}
&\mathit{index}\ 1\ [x] \qquad = x \\
&\mathit{index}\ n\ (_ : xs) = \mathit{index}\ (n - 1)\ xs
\end{aligned}
$$

Again, once an abstraction function is defined and verified, we can enjoy smooth transitions between the two representations, and reap the benefit of having both.

$$\alpha = \mathit{fold}\ (\mathit{leaf} \triangledown (\mathit{fork} \circ \mathit{snd}_{\lambda(x,y) \to \mathit{size}\ x + \mathit{size}\ y}))$$

3.5 Discussion

3.5.1 RINV Expressiveness

The most general constraint on α functions is surjectivity, in order to ensure the existence of the right inverses: valid abstract values are bounded by the actual range of the user-defined α function; invertibility is not guaranteed for abstract values outside this range. In the current proposal, RINV faithfully enforces surjectivity, which explains its restricted

expressiveness compared to the standard pointfree programming framework. An already-mentioned example that shows this difference is the combinator $\triangle$, which executes both of its input functions, and is defined as

$$(\triangle) :: (a \to b) \to (a \to c) \to a \to (b, c)$$
$$(f \triangle g) = \lambda x \to (f\ x, g\ x)$$

Since $f \triangle g$ is generally not surjective, it doesn't have a right inverse, despite the fact that we can easily guard against inconsistent input in the reverse direction as follows.

$$[\![f \triangle g]\!]^{<} = \lambda(a, b) \to \textbf{if } x \texttt{ == } y \textbf{ then } x \textbf{ else } \mathit{error}\ \texttt{"violation"}$$
$$\quad \textbf{where } x = [\![f]\!]^{<}\ a\ ;\ y = [\![g]\!]^{<}\ b$$

Definitions like the one above are known as *weak right inverses* [MMM+07].

Another useful function is *unzip*, which can be defined as a fold.

$$\mathit{unzip} = \mathit{fold}_{\mathit{List}}\ ((\mathit{nil} \triangle \mathit{nil}) \triangledown ((\mathit{cons} \times \mathit{cons}) \circ \mathit{trans}))$$

This definition will be rejected in RINV, since $\mathit{cons} \times \mathit{cons}$ and $\mathit{nil} \triangle \mathit{nil}$ are not jointly surjective. Indeed, *unzip* only produces pairs of lists of equal length.

If a value outside the range is constructed, the integrity of specification level equational reasoning may be corrupted. On the other hand, it is valid to argue that the same invariant assumed for the original datatype prior to the refactoring applies to the specification too. For example, consider a program that requires balanced binary trees. An abstraction function that only produces balanced binary trees is safe if the invariant is correctly preserved in the original program. It remains an open question whether we should allow programmers to take some reasonable responsibilities, or should insist on enforcing control through the language.

We have included *unfold* as a combinator in RINV as the dual of *fold*. Nevertheless, we recognise the fact that there is no combinator in RINV that creates values of a sum type: the *choice* operator introduced for the right-inverse of $\triangledown$ does not preserve surjectivity . An option is to define the bodies of *unfold*s as primitive functions; but this does require the programmers to deal with *Either* types explicitly, something we have tried to avoid.

3.5.2 The Dual Story

In this paper, we have picked the α function to be user-provided; the design of RINV and the subsequent discussion of refactoring is based on this decision. However, this choice is

by no means absolute. One can well imagine a programmer coming up with γ functions first, and a left-invertible language generating the corresponding α functions; this would give the same invertibility property $\alpha \circ \gamma = id$. The promotion condition can be adapted to involve only γ, as in $f_ \circ \mathcal{F}\ \gamma = \mathcal{G}\ \gamma \circ f$. Nevertheless, the crucial computation law and fusion law that form the foundation of the translation and optimization are still derivable; for computation, we have

$$
\begin{array}{ll}
 & \mathcal{G}\ \alpha \circ f_ \circ \mathcal{F}\ \gamma \\
= & \quad \{\, f_ \circ \mathcal{F}\ \gamma \equiv \mathcal{G}\ \gamma \circ f \,\} \\
 & \mathcal{G}\ \alpha \circ \mathcal{G}\ \gamma \circ f \\
= & \quad \{\, \mathcal{G} \text{ respects composition and identity; } \alpha \circ \gamma \equiv id \,\} \\
 & f
\end{array}
$$

and for fusion:

$$
\begin{array}{ll}
 & \mathcal{H}\ \alpha \circ g_ \circ \mathcal{G}\ \gamma \circ \mathcal{G}\ \alpha \circ f_ \circ \mathcal{F}\ \gamma \\
= & \quad \{\, f_ \circ \mathcal{F}\ \gamma \equiv \mathcal{G}\ \gamma \circ f \,\} \\
 & \mathcal{H}\ \alpha \circ g_ \circ \mathcal{G}\ \gamma \circ \mathcal{G}\ \alpha \circ \mathcal{G}\ \gamma \circ f \\
= & \quad \{\, \mathcal{G} \text{ respects composition and identity; } \alpha \circ \gamma \equiv id \,\} \\
 & \mathcal{H}\ \alpha \circ g_ \circ \mathcal{G}\ \gamma \circ f \\
= & \quad \{\, f_ \circ \mathcal{F}\ \gamma \equiv \mathcal{G}\ \gamma \circ f \,\} \\
 & \mathcal{H}\ \alpha \circ g_ \circ f_ \circ \mathcal{F}\ \gamma
\end{array}
$$

If one were to develop a left-invertible language in a similar style to RINV's, we expect that many problems will dualize. A notable difference is that users of the language will have to use *unfold* (as $\triangledown$ does not preserve injectivity), with destruction functions combined together by the backtracking operator *choice*, resulting in a rather unusual programming style. On the other hand, this explicitness, though inconvenient, does offer more control. For example, the programmer is able to determine how a list shall be split in a right-inverse of *app*.

3.5.3 Nested and Overlapping Patterns

Two well-regarded features of pattern matching are the scalability with respect to nesting and the sharing between overlapping patterns. For example, consider a function that sums elements of a list pair-wise:

$$
\begin{aligned}
&\mathit{pairSum}\ \mathit{Nil} &&= \mathit{Nil}\\
&\mathit{pairSum}\ (\mathit{Cons}\ x\ \mathit{Nil}) &&= \mathit{Cons}\ x\ \mathit{Nil}\\
&\mathit{pairSum}\ (\mathit{Cons}\ x\ (\mathit{Cons}\ y\ ys)) &&= \mathit{Cons}\ (x+y)\ (\mathit{pairSum}\ ys)
\end{aligned}
$$

Nested patterns allow simultaneous matching and variable binding to patterns below top level (such as y above), in contrast to the sequential checking of expressions as guards. There is often a degree of sharing between patterns; for example, an input to *pairSum* that, when evaluated, fails to match the first pattern does not need to be evaluated again for subsequent clauses. This is even more important for pattern matching on abstract representations where non-constant computation (i.e., the α function) may be needed. Our proposal supports both features nicely: nested patterns are written exactly the same way as with datatypes; and execution of α functions is done prior to pattern matching and is shared among all the patterns.

3.5.4 Right-invertible vs. Bidirectional

Our system is based on the right-invertibility provided by RINV. Despite the many interesting examples we have seen, there are limitations. Notably, RINV does not keep the source stable: retrieving a value and putting it back without modification may induce changes at source. This has not been a problem with our design, but rules out the possibility of having multiple abstract representations for a single implementation. For example, as shown in [Wad87a], it can be useful to have both *cons*-lists and *snoc*-lists representations available for pattern matching at the same time.

Another omission of rinv is general data-projection functions. We can only discard information that is re-constructible, in another words redundant. At a glance, it may appear that both the above "flaws" will be fixed if a language framework similar to lenses is used; we could use *put* to support stability of source, and recover projected values by extracting them from the copied source. However, in our case the update operations, which are basically arbitrary functions on abstract representations, are far more complicated than those in the view-update problem, which assumes a single type for the view. As a result, an update from type $(\mathit{List}\ a, \mathit{List}\ a)$ to type $\mathit{List}\ a$ would not be allowed as a database update, but is perfectly sensible in programming with abstract representations. If we use *put* on the resulting list, it is never clear what the source input is.

3.6 Further Applications

3.6.1 Pattern Matching for ADTs

The refactoring technique proposed in this paper rewrites selected definitions using pattern matching into ones using primitive operations; the concrete implementations of the primitive operations are not used in reasoning, and thus are not exposed. In this case, a more structured way of organizing the refactored program is with abstract datatypes (ADTs) [LZ74],

As a matter of fact, the data abstraction framework developed in this paper can be used as a way of introducing pattern matching to certain ADTs. For example, we can construct an ADT for queues:

$$
\begin{aligned}
&\textbf{adt}\ \mathit{Queue}\ a = [\,a\,]\\
&\quad \mathit{emptyQ} :: \mathit{Queue}\ a\\
&\quad \mathit{enQ} \quad :: a \to \mathit{Queue}\ a \to \mathit{Queue}\ a\\
&\quad \mathit{deQ} \quad :: \mathit{Queue}\ a \to \mathit{Queue}\ a\\
&\quad \mathit{first} \quad :: \mathit{Queue}\ a \to a\\
&\quad \mathit{isEmpty} :: \mathit{Queue}\ a \to \mathit{Bool}
\end{aligned}
$$

with the following abstract program as specifications

$$
\begin{aligned}
\mathit{emptyQ} &= [\,]\\
\mathit{first} &= \mathit{head}\\
\mathit{isEmpty} &= \mathit{null}\\
\mathit{enQ}\ a\ q &= q +\!\!+ [\,a\,]\\
\mathit{deQ} &= \mathit{tail}
\end{aligned}
$$

This approach is known as *constructive specification* [Hoa72], which explicitly defines the semantics of operations by expressing them in terms of a model. For example, the queue ADT is related to the list model. It is worth emphasising that the list datatype acts only as a model of the ADT: it may suggest but it does not imply a particular implementation. We also note that this constructive approach does not cover all ADTs: for example, sets cannot be fully modeled by an algebraic datatype.

Once an implementation of the ADT is shown correct (by proving the promotion condition), users of the ADT can pattern match on the model when defining non-primitive functions, and reason about them on the level of models. A translation based on the computation law, similar to the one in Section 3.4.2, elaborates the semantics of programs using the ADT.

3.6.2 Stream Fusion

Streams can used as an alternative implementation of lists. Instead of being built as data structures, streams encapsulate operations that can be unfolded to produce stream elements. In [CLS07], Coutts et al. introduce an ADT of streams in their work on fusion optimizations:

$$
\begin{aligned}
&\textbf{data } \mathit{Stream}\ a = \forall s.\mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ s \\
&\textbf{data } \mathit{Step}\ a\ s = \mathit{Done} \\
&\qquad\qquad\quad \mid\ \mathit{Yield}\ a\ s \\
&\qquad\qquad\quad \mid\ \mathit{Skip}\ s
\end{aligned}
$$

To use the stream ADT as an implementation of lists, they have functions *stream* and *unstream* for converting lists to and from streams.

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \to [a] \\
&\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{unfold}\ s \\
&\quad \textbf{where } \mathit{unfold}\ s = \textbf{case } \mathit{next}\ s \textbf{ of} \\
&\qquad\qquad\qquad\qquad \mathit{Done} \quad \to \quad [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Skip}\ s' \quad \to \quad \mathit{unfold}\ s' \\
&\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ s' \to x : \mathit{unfold}\ s' \\
&\mathit{stream} :: [a] \to \mathit{Stream}\ a \\
&\mathit{stream}\ \mathit{xs} = \mathit{Stream}\ \mathit{next}\ \mathit{xs} \\
&\quad \textbf{where } \mathit{next}\ [\,] \quad\ = \mathit{Done} \\
&\qquad\quad\ \ \mathit{next}\ (x : \mathit{xs}) = \mathit{Yield}\ x\ \mathit{xs}
\end{aligned}
$$

The *stream* function is non-recursive, corresponding to the non-recursive definition of *Stream*. The *unstream* function repeatedly calls the access operation of the stream, and produces a list by accumulating the elements this yields. The step *Skip* is unproductive, and therefore does not contribute to expressive power; however, it is crucial for their implementation.

For efficiency, any intermediate stream/unstream conversions must be fused. For example, they want to reduce the expression

$$\mathit{unstream} \circ \mathit{maps}\ f \circ \mathit{stream} \circ \mathit{unstream} \circ \mathit{maps}\ g \circ \mathit{stream}$$

to

$$\mathit{unstream} \circ \mathit{maps}\ f \circ \mathit{maps}\ g \circ \mathit{stream}$$

This would follow from $stream \circ unstream = id$; but this property is not satisfied by their conversion functions, and so their paper leaves open the question of soundness. Nevertheless, the result they want still holds; in fact, it is a consequence of our approach (Theorem 3.6), with *unstream* the abstraction function and *stream* its right inverse.

3.7 Related Work

Efforts to combine data abstraction and pattern matching started two decades ago with Wadler's *views* proposal [Wad87a]; and it is still a hot research topic [BC93, Oka98, Erw96, PGPN96, EP00, Tul00, Jay04, LP07, SNM07, NMN08]. To avoid possible confusion over terminology, we always refer the proposal in [Wad87a] as "Wadler's views".

Wadler's views provide different ways of viewing data than their actual implementations. With a pair of conversion functions, data can be converted *to* and *from* a view. Consider the forward and backward representations of lists:

$$
\begin{array}{l}
\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a) \\
\textbf{view}\ List\ a = Lin \mid Snoc\ (List\ a)\ a \\
\quad to\ Nil \qquad\qquad\qquad\qquad = Lin \\
\quad to\ (Cons\ x\ Nil) \qquad\qquad = Snoc\ Nil\ x \\
\quad to\ (Cons\ x\ (Snoc\ xs\ y)) \quad = Snoc\ (Cons\ x\ xs)\ y \\
\quad from\ Lin \qquad\qquad\qquad = Nil \\
\quad from\ (Snoc\ Nil\ x) \qquad\quad = Cons\ x\ Nil \\
\quad from\ (Snoc\ (Cons\ x\ xs)\ y) = Cons\ x\ (Snoc\ xs\ y)
\end{array}
$$

The **view** clause introduces two new constructors, namely *Lin* and *Snoc*, which may appear in both terms and patterns. The first argument to the view construction *Snoc* refers to the datatype $List\ a$, so a snoclist actually has a conslist as its child. The *to* and $from$ clauses are similar to function definitions. The *to* clause converts a conslist value to a snoclist value, and is used when *Lin* or *Snoc* appear as the outermost constructor in a pattern on the left-hand side of an equation. Conversely, the $from$ clause converts a snoclist into a conslist, when *Lin* or *Snoc* appear in an expression. Note that we are already making use of views in the definition above; for example, *Snoc* appears on the left-hand side of the third *to* clause; matching against this will trigger a recursive invocation of *to*.

Functions can now pattern match on and construct values in either the datatype or one of its views.

$$
\begin{aligned}
&\mathit{last}\ (\mathit{Snoc}\ \mathit{xs}\ x) = x \\
&\mathit{rotLeft}\ (\mathit{Cons}\ x\ \mathit{xs}) \ = \mathit{Snoc}\ \mathit{xs}\ x \\
&\mathit{rotRight}\ (\mathit{Snoc}\ \mathit{xs}\ x) = \mathit{Cons}\ x\ \mathit{xs} \\
&\mathit{rev}\ \mathit{Nil} \qquad\quad = \mathit{Lin} \\
&\mathit{rev}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Snoc}\ (\mathit{rev}\ \mathit{xs})\ x
\end{aligned}
$$

Upon invocation, an argument is converted into the view by the *to* function; after completion of the computation, the result is converted back to the underlying datatype representation.

Just as with our proposal, this semantics can be elaborated by a straightforward translation into ordinary Haskell. First of all, view declarations are translated into data declarations.

$$\textbf{data}\ \mathit{Snoc}\ a = \mathit{Lin} \mid \mathit{Snoc}\ (\mathit{List}\ a)\ a$$

Note that the child of *Snoc* refers to the underlying datatype: view data is typically hybrid (whereas it is homogeneous with our approach). Now the only task is to insert the conversion functions at appropriate places in the program.

$$
\begin{aligned}
&\mathit{last}\ \mathit{xs} = \textbf{case}\ \mathit{to}\ \mathit{xs}\ \textbf{of}\ \mathit{Snoc}\ \mathit{xs}\ x \to x \\
&\mathit{rotLeft}\ \mathit{xs} \ = \textbf{case}\ \mathit{xs}\ \textbf{of}\ \mathit{Cons}\ x\ \mathit{xs} \to \mathit{from}\ (\mathit{Snoc}\ \mathit{xs}\ x) \\
&\mathit{rotRight}\ \mathit{xs} = \textbf{case}\ \mathit{to}\ \mathit{xs}\ \textbf{of}\ \mathit{Snoc}\ \mathit{xs}\ x \to \mathit{Cons}\ x\ \mathit{xs} \\
&\mathit{rev}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of} \\
&\quad \mathit{Nil} \qquad\quad \to \mathit{from}\ \mathit{Lin} \\
&\quad (\mathit{Cons}\ x\ \mathit{xs}) \to \mathit{from}\ (\mathit{Snoc}\ (\mathit{rev}\ \mathit{xs})\ x)
\end{aligned}
$$

In contrast to our approach, Wadler exposes both a datatype and its views to programmers. To support reasoning across the different representations, the conversion clauses are used as axioms. It is expected for a view type to be isomorphic to a subset of its underlying datatype, and for the pair of conversions between the values of the two types to be each other's full inverses. This is certainly restrictive; and Wadler didn't suggest any way to enforce such an invertibility condition. As pointed out by Wadler himself [Wad87a], and followed up by several others [BC93, Oka98], this assumption is risky, and may lead to nasty surprises that threaten soundness of reasoning.

Inspired by Wadler's proposal, our work ties up the loose ends of views by hiding the implementation of selected primitives that are proven correct, and using only the view

(our abstract representation) for pattern matching in user-defined functions. The language RINV for defining conversions guarantees right invertibility, a weaker condition that lifts the isomorphism restriction on abstract and concrete representations. In contrast to Wadler's views, our system only caters for linear refactoring, without having multiple views for the same implementation.

'Safe' variants of views have been proposed before [BC93, Oka98]. To circumvent the problem of equational reasoning, one typically restricts the use of view constructors to patterns, and does not allow them to appear on the right-hand side of a definition. As a result, expressions like *Snoc Lin* 1 become syntactically invalid. Instead, values are only constructed by 'smart constructors', as in *snoc lin* 1. In this setting, equational reasoning has to be conducted on the source level with explict applications of *to*. A major motivation for such a design is to admit views and sources with conversion functions that do not satisfy the invertibility property. In another words, let *Constr* and *constr* be a constructor and its corresponding smart constructor; in general, we have $Constr\ x \not\equiv constr\ x$. This appears to hinder program comprehension, since the very purpose of the convention that the name of a smart constructor differs only by case from its 'dumb' analogue is to suggest the equivalence of the two.

More recently, language designers have started looking into more expressive pattern mechanisms. *Active patterns* [Erw96, PGPN96] and many of their variants [EP00, Tul00, Jay04, LP07, SNM07, PGPN96] go a step further, by embedding computational content into pattern constructions. All the above proposals either explicitly recognise the benefit of using constructors in expressions, or use examples that involve construction of view values on the right-hand sides of function definitions. Nevertheless, none of them are able to support pattern constructors in expressions, due to the inability to reason safely. Knowing that there is an absence of good solutions for supporting constructors in expressions, some work focusses mainly on examples that are primarily data consumers, an escape that is expected to be limited and short-lived. Another common pitfall of active patterns is the difficulty in supporting nested and overlapping patterns, as discussed in Section 3.5.3, because each active pattern is computed and matched independently.

A similar idea that exploits the changing of data representations is the *worker/wrapper* transformation [GH09], for the purpose of replacing an operation with a more efficient one on a different representation. A classic example of it is the transformation of the factorial function on integers (the equivalence of our abstract program) into a more efficient version on unboxed integers (the equivalence of our concrete program) [PJL91]. In contrast to our attempt to systematically guarantee the right-invertibility of the abstraction

functions, the worker/wrapper transformation relies on user-provided proofs. Instead, they focus on the derivation of the concrete program from the syntactic definition of the abstract program; while we only attempt to verify the correctness of user-provided concrete programs with respect to the corresponding abstract programs.

3.8 Conclusion

Algebraic datatypes and pattern matching offer great promise to programmers seeking simple and elegant programming, but the promise turns sour when modular changes are demanded. Our work tackles this long-standing problem by proposing a framework for refactoring programs written with pattern matching into ones with proper encapsulation: programmers are able to selectively re-implement original function definitions into primitive operations, and either rewrite the rest in terms of the primitive ones, or simply leave them unchanged. This migration is completely incremental: executability and proofs through equational reasoning are preserved at all times during the process.

At the heart of our proposal is the framework of data abstraction. When an abstraction function is verified by the promotion condition, the computation law is able to replace abstract function calls with concrete ones. The soundness of such refactoring is based on the right-inverse property of the conversion pairs that bridge the abstract and concrete representations.

Some readers may notice how little we have had to mention the problem of invertibility in Section 3.4, when we illustrate our design of selective refactoring; instead we simply assume it in almost every proof. We think this is a nice showcase for bidirectional programming. Very often invertibility is not the final goal, but part of a solution; bidirectional programming solves it, and solves it so well that we can put our focus on other issues.

Chapter 4

Looking from the Left

If you know yourself but not the enemy,
for every victory gained you will also suffer a defeat.
(Sun Tzu, The Art of War (6th century BC))

4.1 Introduction

Various kinds of program analysis are at the centre of the never-ending quest of detecting program errors and improving run-time performance. Very often, an analysis involves reporting the result back to the programmer in a readable format, a typical example of which is type checking where errors together with their locations and contexts are reported. Consequently, type checking has to be carried out at the earliest stage of compilation, before other transformations deface the source code. This is certainly sub-optimal for compiler writers, where the handful of nice typing rules for a core language explode to tens, even hundreds, to handle various syntactic sugar and extensions. Being an inconvenience for serious compiler implementations with ample resources, this cost is even less acceptable for the very common light-weight approaches such as many domain specific languages (DSLs), where the very reason for building upon a general purpose *host* language is to save the effort of crafting a compiler from scratch. As a direct result, the quality of analysis feedback suffers, which very often reduces to a result in terms of the translated internal code, instead of the source code recognizable to the user. In this case, even the simplest form of error reporting – pointing out the line number in the source code – greatly enhances the programmer's diagnostic experience. As an example, consider an introductory example of Haskell: *quicksort*.

$$
\begin{aligned}
&\mathit{quicksort}\ [\,] \qquad = [\,] \\
&\mathit{quicksort}\ (x : xs) = \mathit{smallerSorted} +\!\!+ [x] +\!\!+ \mathit{biggerSorted} \\
&\quad \textbf{where}\ \mathit{smallerSorted} = \mathit{quicksort}\ [[y] \mid y \leftarrow xs, y \leqslant x] \\
&\qquad\quad \mathit{biggerSorted}\ \ = \mathit{quicksort}\ [[y] \mid y \leftarrow xs, y > x]
\end{aligned}
$$

The list comprehensions are purely syntactic sugar (nothing less than a small scale DSL for the more maths-minded), and are translated into applications of the higher-order function *concatMap*. For example, according to the Haskell report [PJ03] the expression $[[y] \mid y \leftarrow xs, y \leqslant x]$ is translated into

$$
\begin{aligned}
&\textbf{let}\ f\ y = \textbf{if}\ y \leqslant x\ \textbf{then}\ [[y]]\ \textbf{else}\ [\,] \\
&\quad f\ _ = [\,] \\
&\textbf{in}\ \mathit{concatMap}\ f\ xs
\end{aligned}
$$

Astute readers may have already noticed that the program above is actually faulty: the output ys of the list comprehensions are erroneously written as $[y]$. Consequently, the translated program is type-incorrect as well. However, complaining that an infinite

type is being constructed in f's definition is not very helpful to a programmer who only sees the list comprehensions.

The same story goes beyond type checking; other analyses that involve feeding back, such as reachability and termination checks, suffer the same fate. Moreover, for incomplete analyses like the ones just mentioned, it is often profitable to apply them to simpler core languages for better results [MR09].

The challenge is evident here: on the one hand we desire simple and precise analyses on transformed code, while on the other hand the practical need for reporting results fixes the analyses to the source code. We approach this problem with a novel idea: bidirectionalizing the program transformations.

Before setting out on the task of finding a suitable bidirectional technique, we need to understand the language requirements of the application. First of all, as program transformations are very often complicated, language expressiveness is a key requirement here. Thus, bidirectional laws, which do not impose stringent injective or surjective restrictions, are more suitable than invertibility to be the bidirectional property here. We also notice that the bidirectional transformation here is "one-trip": once the analysis result (updated view) is mapped back to the source code (source), the process is finished. In another words, we have a changeable, but not observable, view, which suggests that consistency is probably optional since it is for the consistency of the view. To confirm this speculation, we further observed that only some "auxiliary" information in the view (namely the analysis results possibly integrated with the code as annotations), are changeable; and the putting back of those information depends on the semantics of the program being handled, and is best left to the programmer. For example, if we discover an error in the definition of f in the above example, whether to blame the complete list comprehension in the source or just the output part is rather a subjective choice; and when fed to the retrieval function again, the different choices may not produce exactly the same view. In this case, rigidly enforcing consistency may take away much-needed flexibility without much benefit. We will see more discussion on this in the sequel. On the other hand, acceptability – which keeps the integrity of the source – is crucially important, because any interpretation of the auxiliary information is only possible if it is attached to the correct source.

Last but not least, the language of the bidirectional system needs to support the tools of the trade. In Haskell, this means that pattern matching, on very often deeply nested syntax trees, is a must-have.

Based on the above requirements, we choose to use the constant complement approach to tackle the problem, largely following from the technique in [MHN+07, Section 2.1.2.1]. As we will see, the constant complement approach allows us to keep the pointwise style of programming, commonly used in program transformations, and effectively bidirectionalize the code. The drawback of the original technique in [MHN+07] is updatability: to support decidable update checking, the language for retrieval functions is severely restricted. In our case, we take advantage of the update pattern of our application and extend the system in [MHN+07]. As a result, our proposed language is more practical and handles the intended update safely.

4.2 Bidirectionalization Transformation

In this section, we present our bidirectionalization technique, centred around the construction of complement values. We assume total retrieval functions, and fully-defined source values.

A complement keeps as data whatever information was lost during the retrieval process, so that when it is combined with the view, the two uniquely determine the source. In [MHN+07], the concept is extended to functions. Suppose a source type S, view type V, and complement type C. For a retrieval function $f :: S \to V$, a *complement function* $f^\bullet :: S \to C$ constructs a complement value from a source. Tupling it with the retrieval function, we obtain an injective function $f \vartriangle f^\bullet :: S \to (V, C)$, which is ready for inversion.

As a simple example, for the retrieval function $fst :: (a, b) \to a$, the function $snd :: (a, b) \to b$ is a complement function, preserving the second element of the input pair. Another example is a retrieval function that flattens trees into lists; the complement function will need to preserve the exact shape of the tree, but may leave holes for leaf values.

The complement functions are usually not unique; for example, in addition to *snd*, both $id :: a \to a$ and $swap :: (a, b) \to (b, a)$ are complement functions for *fst*. There are multiple ways of encoding the shape of a tree as well.

After obtaining the injective tupling of retrieval and complement functions, reversing it for a putback function becomes straightforward. We illustrate the complete process through an example before formalizing it. As a notational convention, we superscript complement functions by '•', tupled functions by '⩤',

4.2.1 An Example

We begin with the familiar binary tree and list datatypes.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \\
&\qquad\qquad \mid\ \mathit{Leaf}\ a \\
&\qquad\qquad \mid\ \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)
\end{aligned}
$$

Consider the scenario that a binary tree is used to store data, due to its flexibility and potential support for efficient access; and at the same time, a list-like interface offers straightforward manipulation and display. We could use the following retrieval function to transform tree sources into list views.

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Tree}\ a \to \mathit{List}\ a \\
&\mathit{flatten}\ \mathit{Empty} = \mathit{Nil} \\
&\mathit{flatten}\ (\mathit{Leaf}\ x) = \mathit{Cons}\ x\ \mathit{Nil} \\
&\mathit{flatten}\ (\mathit{Branch}\ t_1\ t_2) = l \\
&\quad \textbf{where}\ l_1 = \mathit{flatten}\ t_1 \\
&\qquad\quad\ \ l_2 = \mathit{flatten}\ t_2 \\
&\qquad\quad\ \ l\ = \mathit{append}\ (l_1, l_2)
\end{aligned}
$$

The list concatenation function *append* is defined as follows.

$$
\begin{aligned}
&\mathit{append} :: (\mathit{List}\ a, \mathit{List}\ a) \to \mathit{List}\ a \\
&\mathit{append}\ (\mathit{Nil}, ys) = ys \\
&\mathit{append}\ (\mathit{Cons}\ x\ xs, ys) = \mathit{Cons}\ x\ xys \\
&\quad \textbf{where}\ xys = \mathit{append}\ (xs, ys)
\end{aligned}
$$

Note that no leaf labels are lost during this retrieval transformation, only the structure of the tree. The job of the putback function is to restore the shape of the tree and fill in possibly modified labels.

The basic idea the approach is to keep a history of the control flow of an execution as a complement value, which combined with the output of the execution allows us to trace the execution backwards to reproduce the input. Our bidirectionalization process firstly generates a complement function.

$$
\begin{aligned}
&\textbf{data}\ \mathit{FlatCompl} = \mathit{CaseEy} \\
&\qquad\qquad\qquad \mid\ \mathit{CaseLf}
\end{aligned}
$$

$$
\begin{aligned}
&\quad \mid \mathit{CaseBr}\ \mathit{FlatCompl}\ \mathit{FlatCompl}\ \mathit{AppCompl}\\
&\mathit{flatten}^\bullet :: \mathit{Tree}\ a \to \mathit{FlatCompl}\\
&\mathit{flatten}^\bullet\ \mathit{Empty} \qquad\quad = \mathit{CaseEy}\\
&\mathit{flatten}^\bullet\ (\mathit{Leaf}\ x) \qquad = \mathit{CaseLf}\\
&\mathit{flatten}^\bullet\ (\mathit{Branch}\ t_1\ t_2) = \mathit{CaseBr}\ (\mathit{flatten}^\bullet\ t_1)\ (\mathit{flatten}^\bullet\ t_2)\ (\mathit{append}^\bullet\ (l_1, l_2))\\
&\quad \textbf{where}\ l_1 = \mathit{flatten}\ t_1\\
&\qquad\quad\ \ l_2 = \mathit{flatten}\ t_2\\
&\qquad\quad\ \ l\ \ = \mathit{append}\ (l_1, l_2)
\end{aligned}
$$

We introduce a new complement datatype *FlatCompl*, with one constructor for each clause of *flatten*'s definition; the complement function $\mathit{flatten}^\bullet$ maps from clauses to constructors. Recursive calls in *flatten* correspond to recursive substructures in *FlatCompl*; calls to other functions entail additional fields for the corresponding constructors, recording relevant complements. So the first two clauses of *flatten* (with no function calls on the right-hand side) correspond to constructors with no arguments; but the third clause has two recursive calls (hence two substructures) and one nested call to *append* (hence a field for *append*'s complement type *AppCompl*).

$$
\begin{aligned}
&\textbf{data}\ \mathit{AppCompl} = \mathit{CaseNil} \mid \mathit{CaseCons}\ \mathit{AppCompl}\\
&\mathit{append}^\bullet :: (\mathit{List}\ a, \mathit{List}\ a) \to \mathit{AppCompl}\\
&\mathit{append}^\bullet\ (\mathit{Nil}, ys) \qquad\quad = \mathit{CaseNil}\\
&\mathit{append}^\bullet\ (\mathit{Cons}\ x\ xs, ys) = \mathit{CaseCons}\ (\mathit{append}^\bullet\ (xs, ys))\\
&\quad \textbf{where}\ \mathit{xys} = \mathit{append}\ (xs, ys)
\end{aligned}
$$

Basically, an *AppCompl* value encodes the length of a list being deconstructed; for example, *CaseCons* (*CaseCons* (*CaseCons CaseNil*)) represents a list of length three. Given that *AppCompl* is isomorphic to natural numbers, to avoid verbose symbols, we use numbers to represent *AppCompl* values; for example, *CaseCons* (*CaseCons* (*CaseCons CaseNil*)) is written as *Three*.

These complement functions faithfully record the decomposition of the input. For example, applying $\mathit{flatten}^\bullet$ to a tree

$$
\begin{aligned}
&\mathit{Branch}\ (\mathit{Branch}\ (\mathit{Leaf}\ \texttt{'a'})\\
&\qquad\qquad\quad (\mathit{Branch}\ (\mathit{Leaf}\ \texttt{'d'})\ (\mathit{Leaf}\ \texttt{'b'})))\\
&\qquad (\mathit{Leaf}\ \texttt{'c'})
\end{aligned}
$$

gives the complementary structure

$$
\begin{array}{l}
\mathit{CaseBr}\ (\mathit{CaseBr}\ \mathit{CaseLf} \\
\qquad\qquad (\mathit{CaseBr}\ \mathit{CaseLf}\ \mathit{CaseLf}\ \mathit{One})\ \mathit{One}) \\
\qquad \mathit{CaseLf}\ \mathit{Three}
\end{array}
$$

For each branch, there is an *AppCompl* value which records the size of the left-subtree.

When tupled with the retrieval functions, we obtain injective functions $\mathit{flatten}^{\gtrdot}$ and $\mathit{append}^{\gtrdot}$: the element list and the shape of the tree uniquely determine the input tree.

$$
\begin{array}{l}
\mathit{flatten}^{\gtrdot} :: \mathit{Tree}\ a \to (\mathit{List}\ a, \mathit{FlatCompl}) \\
\mathit{flatten}^{\gtrdot}\ \mathit{Empty} \qquad\qquad = (\mathit{Nil}, \mathit{CaseEy}) \\
\mathit{flatten}^{\gtrdot}\ (\mathit{Leaf}\ x) \qquad\quad = (\mathit{Cons}\ x\ \mathit{Nil}, \mathit{CaseLf}) \\
\mathit{flatten}^{\gtrdot}\ (\mathit{Branch}\ t_1\ t_2) \quad = (l, \mathit{CaseBr}\ d_1\ d_2\ c) \\
\quad \textbf{where}\ (l_1, d_1) = \mathit{flatten}^{\gtrdot}\ t_1 \\
\qquad\qquad (l_2, d_2) = \mathit{flatten}^{\gtrdot}\ t_2 \\
\qquad\qquad (l, c) \quad = \mathit{append}^{\gtrdot}\ (l_1, l_2) \\
\mathit{append}^{\gtrdot} :: (\mathit{List}\ a, \mathit{List}\ a) \to (\mathit{List}\ a, \mathit{AppCompl}) \\
\mathit{append}^{\gtrdot}\ (\mathit{Nil}, ys) \qquad\quad = (ys, \mathit{CaseNil}) \\
\mathit{append}^{\gtrdot}\ (\mathit{Cons}\ x\ xs, ys) = (\mathit{Cons}\ x\ xys, \mathit{CaseCons}\ c) \\
\quad \textbf{where}\ (xys, c) = \mathit{append}^{\gtrdot}\ (xs, ys)
\end{array}
$$

Note that complement function calls on the right-hand side are subsumed by tupled function calls in the **where** clauses. Now, the task of creating a putback function reduces to swapping patterns between the two sides of the equations.

$$
\begin{array}{l}
\mathit{flatten}^{\lessdot} :: (\mathit{List}\ a, \mathit{FlatCompl}) \to \mathit{Tree}\ a \\
\mathit{flatten}^{\lessdot}\ (\mathit{Nil}, \mathit{CaseEy}) \qquad\quad = \mathit{Empty} \\
\mathit{flatten}^{\lessdot}\ (\mathit{Cons}\ x\ \mathit{Nil}, \mathit{CaseLf}) = (\mathit{Leaf}\ x) \\
\mathit{flatten}^{\lessdot}\ (l, \mathit{CaseBr}\ d_1\ d_2\ c) \quad = (\mathit{Branch}\ t_1\ t_2) \\
\quad \textbf{where}\ t_1 = \mathit{flatten}^{\lessdot}\ (l_1, d_1) \\
\qquad\qquad t_2 = \mathit{flatten}^{\lessdot}\ (l_2, d_2) \\
\qquad\qquad (l_1, l_2) = \mathit{append}^{\lessdot}\ (l, c) \\
\mathit{append}^{\lessdot} :: (\mathit{List}\ a, \mathit{AppCompl}) \to (\mathit{List}\ a, \mathit{List}\ a) \\
\mathit{append}^{\lessdot}\ (ys, \mathit{CaseNil}) \qquad\qquad = (\mathit{Nil}, ys) \\
\mathit{append}^{\lessdot}\ (\mathit{Cons}\ x\ xys, \mathit{CaseCons}\ c) = (\mathit{Cons}\ x\ xs, ys) \\
\quad \textbf{where}\ (xs, ys) = \mathit{append}^{\lessdot}\ (xys, c)
\end{array}
$$

The complement information serves as a roadmap for the putback function $\mathit{flatten}^{\lessdot}$: given a list of appropriate length, $\mathit{flatten}^{\lessdot}$ can determine exactly where to split the list to form

subtrees and place leaf elements; it constructs a source value identical in structure to the original.

4.3 The Algorithm

We now proceed to formalise the process illustrated above. To start with, we introduce the miniature first-order language in which our bidirectional system is defined. The syntax is presented below.

$$\begin{array}{ll} \text{Prog} & \pi ::= \overline{d} \\ \text{Decl} & d ::= f\ p \mid f\ e = e\ \textbf{where}\ \overline{p = f\ e} \\ \text{Exp} & e ::= C\ e \mid x \\ \text{Patt} & p ::= x \mid C\ p \end{array}$$

(The | is syntax for a guarded definition; the **|** is metasyntax for choice in the grammar.) We write $\overline{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. declarations, variables, and so on), and $fv(o)$ as the free variables in o. When used together, o and $\overline{o}$ denote unrelated objects. As a notational convention, C, f and x range over data constructors, function names and variables respectively.

A program in the language is a sequence of function declarations. The expressions of the language include variables and data constructions; function applications only appear in **where** clauses and pattern guards; nesting of function calls are expressed through nesting of **where** clauses. We also omit conditionals from expressions, encoding them with pattern guards in function declarations, and assume non-overlapping patterns. The above choices do not limit expressiveness, but effectively process function definitions into a canonical form more convenient for manipulation. We do not include types, as our primary concern is the syntactic transformation of functions, though we may use type signatures in examples to improve readability.

We omit a fixed operational semantics for the language, since it is standard. The bidirectionalization process progresses in three steps, and produces a putback function in the same language. We look into each of the steps in more detail below.

4.3.1 Constructing the Complement Function

Definition 4.1 (Complement Function) *A function $f^\bullet :: S \to C$ is a complement function to a retrieval function $f :: S \to V$, if the tupling of them $f \vartriangle f^\bullet :: S \to (V, C)$ is injective.*

The syntactic translation that generates a complement function, from and for each retrieval functions, is the following:

$$\frac{\overline{z} = fv(\overline{p}, p) \setminus fv(e, \overline{ew}) \quad C \textit{ fresh}}{\begin{array}{c} f\ p \mid g = e\ \textbf{where}\ \overline{pw = h\ ew} \ \rightsquigarrow\ f^\bullet\ p \mid g = C\ \overline{h^\bullet\ ew}\ \overline{z}\ \textbf{where}\ \overline{pw = h\ ew} \\ C :: \overline{\tau_w} \to \overline{\tau_z} \to \tau_{f^\bullet} \end{array}}$$

The side conditions are above the line, with the complement function generation beneath it. We generate complement functions indiscriminately for all top-level declared functions. The body of a function definition is turned into a complement value expression, which is constructed by applying a fresh data constructor C to the complement value expressions resulting from replacing the function calls in the **where** clauses to their complement conterparts. Note that each **where** clause contains exactly one function application, and produces one complement value. Intuitively, if we consider a complement as the memoization of a function computation, it needs to record the subcomputations (function calls) originating from that function and all locally bound term variables that are unused in the right-hand side of the clause. The **where** clauses themselves are copied over to the complement function because variables bound in $\overline{pw}$ may be used in $\overline{h^\bullet\ ew}$ (see the *flatten* function in the previous section as an example). The fresh constructor C belongs to the complement datatype $\tau_{f^\bullet}$, created for each retrieval function. The constructor takes parameters of types $\overline{\tau_w}$ and $\overline{\tau_z}$, which represents the types of $\overline{h^\bullet\ ew}$ and $\overline{z}$ respectively.

Theorem 4.2 *Given a function f such that $f \ \rightsquigarrow\ f^\bullet$, then $f^\bullet$ is a complement function to f.*

Proof. Follows directly from Corollary 4.4. □

Lemma 4.3 *Given a function f such that $f \ \rightsquigarrow\ f^\bullet$, for any y and z*

$$y \ \neq\ z \Rightarrow (f\ y, f^\bullet\ y) \ \neq\ (f\ z, f^\bullet\ z)$$

Proof. Let r and r' be the clauses of f that are invoked by applying it to y and z respectively.

If $r \neq r'$, the proof is done, we have $f^\bullet\ y \neq f^\bullet\ z$ because the complement constructor C is different for each clause.

If $r = r'$, we prove by induction on the nesting of function-calls on the right-hand side of r.

Base case: there is no function call on the right-hand of r. Let r and its corresponding complement-function clause be defined as

$$f\ p \mid g = e$$
$$f^\bullet\ p \mid g = C\ \overline{z}$$

By the translation rule, we have $fv(p) = fv((e, C\ \overline{z}))$. We use the notation $[p \mapsto x]$ to represent the substitution arises from matching pattern p to term x. Given $x \neq y$ and injectivity of constructors, we have

$$(e, C\ \overline{z})[p \mapsto y] \neq (e, C\ \overline{z})[p \mapsto z]$$

which is equivalent to $(f\ y, f^\bullet\ y) \neq (f\ z, f^\bullet\ z)$.

Inductive Step: Let r be defined as

$$f\ p \mid g = e\ \textbf{where}\ \overline{pw = h\ ew}$$

and is translated to

$$f^\bullet\ p \mid g = C\ \overline{h^\bullet\ ew}\ \overline{z}\ \textbf{where}\ \overline{pw = h\ ew}$$

The flattened nested function application is rather awkward for expressing the induction hypothesis. We invent a couple of auxiliary function to encapsulate the **where** clauses:

$$f1\ p \mid g = \overline{pw}\ \textbf{where}\ \overline{pw = h\ ew}$$

$$f1^\bullet\ p \mid g = \overline{h^\bullet\ ew}\ \textbf{where}\ \overline{pw = h\ ew}$$

With $f1$, we can rewrite f into

$$f\ p \mid g = D\ (f1\ p)\ p$$

with a constructor D that constructs the expression e from $f1\ p$ and p.
Now assuming

$$y \neq z \Rightarrow (f1\ y, f1^\bullet\ y) \neq (f1\ z, f1^\bullet\ z)$$

we want to prove

$$y \neq z \Rightarrow (f\ y, f^\bullet\ y) \neq (f\ z, f^\bullet\ z)$$

Given $y \neq z$,

- $f1^\bullet\ y \neq f1^\bullet\ z$, we have $f^\bullet\ y \neq f^\bullet\ z$ due to the injectivity of C.
- $f1\ y \neq f1\ z$, we have $f^\bullet\ y \neq f^\bullet\ z$ due to the injectivity of D.

As a remark, we don't need to consider the guards in the above proof because their roles are completely captured by the choice of clauses.

□

An injectivity property of $f \vartriangle f^\bullet$ immediately follows from the above lemma.

Corollary 4.4 *Given a function f such that $f \rightsquigarrow f^\bullet$, we have that $f \vartriangle f^\bullet$ is injective.*

□

4.3.2 Tupling

The second step of bidirectionalization is to tuple the retrieval function with its complement to create an injective tupled function. For this purpose, we could simply use the function $f \vartriangle f^\bullet$; but this will take twice as long to execute. As we saw in the previous section, the generated complement function has a similar syntactical structure congruent to the retrieval function, so we can straightforwardly combine the right-hand sides of both functions into a tuple and replace function calls in **where** clauses with their tupled version.

$$\underbrace{f\ p \mid g = e\ \textbf{where}\ \overline{p = h\ e} \qquad f^\bullet\ p \mid g = C_f\ \overline{(h^\bullet\ e)}\ \overline{z}\ \textbf{where}\ \overline{p = h\ e}}$$
$$f^\rhd\ p \mid g = (e, C_f\ \overline{c}\ \overline{z})\ \textbf{where}\ \overline{(p, c) = h^\rhd\ e}$$

The complement function applications $\overline{h^\bullet\ e}$ on the right-hand side of $f^\bullet$ are subsumed by the tupled function applications $\overline{h^\rhd\ e}$ in the **where** clauses on the right-hand side of $f^\rhd$, the return values of which are bound to the freshly generated variable patterns $\overline{c}$. Thus, we use $\overline{c}$ as the arguments of constructor C_f in $f^\rhd$. The notation $\overline{c}$ is overloaded for both syntactically identical terms and patterns.

4.3.3 Inverting the Tupled Function

The putback function can be constructed by directly swapping the pattern and the body of the tupled function. The function applications in the **where** clauses are transformed in a similar way: arguments and results are swapped, and the guard stays, since the putback function is only defined when the guard condition remains satisfied. The final product of bidirectionalization is as follows:

$$f^{<}\ (e, C_f\ \overline{c}\ \overline{z}) \mid g = p \textbf{ where } \overline{e = h^{<}\ (p, \overline{c})}$$

Again, we overload syntactic notations for both terms and patterns. Note that since there is exactly one function application in each **where** clause body, e and $\overline{e}$ do not contain any function applications and can be used as patterns syntactically.

4.3.3.1 Multiple Use of Variables

There is a subtlety here concerning the above inverting step (but not the previous two): if a bound variable is used more than once on the right-hand side of a retrieval function, the pattern of the inverted function fails to be linear. Consider a simple example,

$$\mathit{dup}\ x = (x, x)$$

Inverting it gives us

$$\mathit{dup}^{<}\ (x, x) = x$$

which is not accepted in Haskell. Various solutions have been proposed for this problem. An obvious one is to disallow multiple use of variables altogether [MHN+07]; functions satisfying such a restriction are called *affine* or *linear*. In systems where duplication plays an important role [MHT04b, MHT04a, HMT04], the non-linear pattern is made linear, and the putback function is only defined if the values of the duplicated variables agree.

$$\mathit{dup}^{<}\ (x, y) \mid x \equiv y = x \quad \text{-- or y}$$

Another possibility is to relax the correctness condition by allowing one value to be chosen if they don't all agree. For example, we could ignore one copy.

$$\mathit{dup}^{<}\ (x, y) = x$$

Duplication of variables never invalidates the injectivity property of the tupled functions, which acceptability of our bidirectional system is based on. But the choices of

dealing with them do affect the consistency property. For example, the last definition of $dup^<$ above is not consistent because the ignored y will never not be recovered by a retrieval transformation. However, compared to the first two approaches above that satisfy consistency, the last one allows maximum expressiveness: there is no restriction on the retrieval function and the putback function is total (perfect updatability). In our application of type error reporting, expressiveness out weights consistency as a slightly confusing error report is nevertheless useful, but probably not an over-restrictive language for writing transformations.

For this reason, in this chapter we will focus only on acceptability and updatability. This means we won't be able to formally express how well an error is reported back to the source. However, as we will see shortly, the required user-input in making sure the system being useful is minimum.

4.3.4 Properties of Bidirectionalization

Theorem 4.5 (Acceptability) *Given a retrieval function f*

$$f^< \circ f^> = id$$

Proof. Follow directly from Theorem 4.2, stating $f \triangle f^\bullet$ being injective, and the definition of $f^<$. □

In addition to the bidirectional property above, updatability is another concern. As suggested by the name of the technique, complement values must remain constant, which limits changes to the views. For example, consider the function *append* in Section 5.1.1, where the complement records the length of the first source list; if any updates reduce the view to a shorter list, the putback function $append^<$ will fail due to non-exhaustive patterns.

In this respect, not all complements are equal. For example, a complement that records the lengths of both source lists for *append* accommodates fewer view updates than one that only records the length of the first list. As a result, it is usually considered desirable to "collapse" a complement, so that the possibility of conflicts between it and the updated view is reduced. Nevertheless, in general such conflicts cannot be completely eliminated; it becomes necessary to detect an "invalid" update before the putback function fails. Achieving this involves in-depth understanding of the complement generation process, something not expected from users of bidirectional languages. In [MHN+07], the system performs a range analysis of the tupled function, and automatically rejects any

updates that cause the view/complement pair to be outside of the range. However the analysis is only decidable for a small set of functions, namely *treeless* functions [Wad88], and there is no easy way for users to understand the reasons behind a rejection, let alone to find a remedy.

We look at the issue from a different perspective. Instead of complicated complement reduction and function range analysis, we lift any restrictions on the bidirectional language, and only allow changes to element values of parametric structures. This approach is similar to the semantic bidirectionalization proposal [Voi09, Section 2.1.2.2]. Before going into a more detailed discussion, let's look at another example.

4.3.5 Projection of Data

Consider a filtering function that removes negative numbers from an integer list. The filtered list can be seen as a view derived from the original source.

$$
\begin{aligned}
&\mathit{filter} :: \mathit{List}\ \mathit{Int} \to \mathit{List}\ \mathit{Int} \\
&\mathit{filter}\ \mathit{Nil} && = \mathit{Nil} \\
&\mathit{filter}\ (\mathit{Cons}\ x\ xs) \mid x \geqslant 0 && = \mathit{Cons}\ x\ ys \\
&\qquad && \quad \textbf{where}\ ys = \mathit{filter}\ xs \\
&\mathit{filter}\ (\mathit{Cons}\ x\ xs) && = ys \\
&\qquad && \quad \textbf{where}\ ys = \mathit{filter}\ xs
\end{aligned}
$$

This time, the function does discard concrete elements of the source, so these need to be preserved in the complement. As before, there is one constructor of the complement for each clause of the transformation; in addition to the fields for recursive calls, there is a field for the discarded variables x.

$$
\begin{aligned}
&\textbf{data}\ F = F_1 \\
&\qquad \mid\ F_2\ (F\ \mathit{Int}) \\
&\qquad \mid\ F_3\ (F\ \mathit{Int})\ \mathit{Int} \\
&\mathit{filter}^\bullet :: \mathit{List}\ \mathit{Int} \to F \\
&\mathit{filter}^\bullet\ \mathit{Nil} && = && F_1 \\
&\mathit{filter}^\bullet\ (\mathit{Cons}\ x\ xs) \mid x \geqslant 0 && = && F_2\ (\mathit{filter}^\bullet\ xs) \\
&\qquad && && \quad \textbf{where}\ ys = \mathit{filter}\ xs \\
&\mathit{filter}^\bullet\ (\mathit{Cons}\ x\ xs) && = && F_3\ (\mathit{filter}^\bullet\ xs)\ x \\
&\qquad && && \quad \textbf{where}\ ys = \mathit{filter}\ xs
\end{aligned}
$$

Once again, the tupling is straightforward:

$$
\begin{aligned}
&\mathit{filter}^{\geqslant} :: \mathit{List}\ a \to (\mathit{List}\ a, F\ a) \\
&\mathit{filter}^{\geqslant}\ \mathit{Nil} &&= (\mathit{Nil}, F_1) \\
&\mathit{filter}^{\geqslant}\ (\mathit{Cons}\ x\ xs) \mid x \geqslant 0 &&= (\mathit{Cons}\ x\ ys, F_2\ f) \\
&\qquad\qquad &&\quad \mathbf{where}\ (ys, f) = \mathit{filter}^{\geqslant}\ xs \\
&\mathit{filter}^{\geqslant}\ (\mathit{Cons}\ x\ xs) &&= (ys, F_3\ f\ x) \\
&\qquad\qquad &&\quad \mathbf{where}\ (ys, f) = \mathit{filter}^{\geqslant}\ xs
\end{aligned}
$$

as is the inversion:

$$
\begin{aligned}
&\mathit{filter}^{<} :: (\mathit{List}\ a, F\ a) \to \mathit{List}\ a \\
&\mathit{filter}^{<}\ (\mathit{Nil}, F_1) &&= \mathit{Nil} \\
&\mathit{filter}^{<}\ (\mathit{Cons}\ x\ ys, F_2\ f) \mid x \geqslant 0 &&= \mathit{Cons}\ x\ xs \\
&\qquad\qquad &&\quad \mathbf{where}\ xs = \mathit{filter}^{<}\ (ys, f) \\
&\mathit{filter}^{<}\ (ys, F_3\ f\ x) &&= \mathit{Cons}\ x\ xs \\
&\qquad\qquad &&\quad \mathbf{where}\ xs = \mathit{filter}^{<}\ (ys, f)
\end{aligned}
$$

The reconstruction of the source list is again interesting. The complement information not only records each position from which an element is removed (by differences in constructors), but also provides the exact values to be inserted back (by storing those values as arguments to constructors). If an element in the view is changed to a negative value, the putback execution will fail, because the complement constructor F_2 conflicts with any element x such that $x \geqslant 0$ is false.

4.3.6 The Anatomy of Complements

In contrast to the *flatten* example, the complement of *filter* not only needs to record the path of execution, but also the dropped elements. We call the part of the complement that stores dropped values the *data complement*, separating it form the *structure complement* that reflects the execution path. Data complement and structure complement often mingle together as in the case of *filter*: the structure part, denoted by the constructors and the recursive occurrences of type F, encodes the execution path; and the data part, denoted by the parameters of type *Int*, stores discarded data.

Since data complements never appear on the right-hand side (including the pattern guard) of an equation, they can never be violated by an update if its companion structure complement is not violated. In term of updatability, where the violation of complements

is the primary concern, it is the structure complement that matters. In other words, for values in a source that do not contribute to the structure complement, their appearances (if they do appear) in the view can be changed without violating the complement. Finding such values involves non-trivial scrutiny of the semantics of the transformation. But there are times that we do get things for free! For a retrieval function of parametrically polymorphic type $S\ a \to V\ a$, *free theorems* [Wad89] guarantee that values corresponding to the polymorphic type variables a do not affect the execution and, thus, the structure complement.

Definition 4.6 *Two containers $x :: S\ t$ and $y :: S\ t$ are structurally equal, denoted by $\simeq$, if $fmap\ (const\ ())\ x = fmap\ (const\ ())\ y$.*

Changes only the elements of a parametric view can always be put back to the source.

Fact 4.7 *Given a polymorphic retrieval function $f :: S\ a \to V\ a$, the execution of $f^{<}\ (v, f^{\bullet}\ s)$ always succeeds for any $v \simeq f\ s$.*

Looking back to our examples, for polymorphic function *flatten*, applying *flatten* on a view with different element values is always defined, whereas the same is not true for non-polymorphic function *filter*, since the element values decide the structure complement. This result is backed up by the updatability result of semantic bidirectionalization [Voi09]. One reviewer of our paper that combines syntactic and semantic bidirectionalization for better updatability [VHMW10] observed from examples that when used independently the syntactic approach always outperforms the semantic approach in term of updatability. The development here confirms this observation.

4.4 Program Analysis Result Reporting

We are now ready to apply the above bidirectional system. Given a source program, we firstly transform it, and then perform an analysis on the translated program. Any results are put back to appropriate contexts in the source program through putback execution of the transformation. In the sequel, we look into the details of different types of typical program transformation with examples, and demonstrate the use of bidirectionalization for improved result reporting.

4.4.1 Annotated Expressions

To store the results of program analysis, we assume annotated ASTs. As a result, every expression additionally carries a field storing the extra information. For example, consider a toy expression language.

$$
\begin{aligned}
&\textbf{data } \mathit{Exp} = \mathit{Var\ String} \\
&\qquad\quad \mid \mathit{INT\ Int} \\
&\qquad\quad \mid \mathit{Lam\ String\ Exp} \\
&\qquad\quad \mid \mathit{App\ Exp\ Exp}
\end{aligned}
$$

Annotating it involves building a mutually recursive pair of types.

$$
\begin{aligned}
&\textbf{type } \mathit{ExpA\ a} = (a, \mathit{Exp'}) \\
&\textbf{data } \mathit{Exp'\ a} = \mathit{Var\ String} \\
&\qquad\quad \mid \mathit{INT\ Int} \\
&\qquad\quad \mid \mathit{Lam\ String\ (ExpA\ a)} \\
&\qquad\quad \mid \mathit{App\ ExpA\ (ExpA\ a)}
\end{aligned}
$$

We deliberately make the type of annotations polymorphic, not only to make them general holders of any analysis results, but also to high-light the fact that annotations will not be inspected by the transformations. From the result of the previous section, given a transformation of type $\mathit{ExpA\ a} \rightarrow \mathit{ExpA\ a}$, a program analyser can freely update the annotations without worrying about whether the putback function is sufficiently defined.

4.4.2 Bidirectionalizing Transformations

We assume Haskell syntax with annotated expressions. To clarify presentation, we use the `teletype` font for expressions as datatypes to distinguish from the Haskell code that manipulates them. We also denote annotations as superscripts. From time to time, we omit irrelevant annotations for brevity.

4.4.2.1 Rearranging expressions

Very often, program transformations involve rearranging expressions. Consider the following implementation that encodes a few humble optimizations (out of many more) used in the GHC [JS94].

$$
\begin{array}{lll}
optz & \texttt{((let b in e}^{h}\texttt{)}^{i}\ \texttt{arg}^{j}\texttt{)}^{k} & = \texttt{(let b in (e}^{h}\ \texttt{arg}^{j}\texttt{)}^{k}\texttt{)}^{k} \\
optz & \texttt{(let x = (let b in e1}^{h}\texttt{)}^{i}\texttt{in e2}^{j}\texttt{)}^{k} & = \texttt{(let b in (let x = e1}^{h}\ \texttt{in e2}^{j}\texttt{)}^{k}\texttt{)}^{k} \\
optz & \texttt{(let v = e1}^{h}\ \texttt{in e2}^{i}\texttt{)}^{j} & = \texttt{(case e1}^{h}\ \texttt{of v} \rightarrow \texttt{e2}^{i}\texttt{)}^{j}
\end{array}
$$

Function *optz* is a Haskell function that manipulates its input, being the language that is manipulated. Each clause above represents a particular optimization of `let`. The first one flows application inwards to the body, which opens up potential possibilities of further beta reductions. The second clause normalizes a deeply nested `let`, by moving a `let` in the binding out to the top level. Possible name capturing is certainly a concern here; GHC avoids it by aggressively renaming to fresh variables (a step which is omitted here, but will be addressed later). The third clause converts a `let` into a `case` when the `let` is strict, to save some heap allocation for the bound variable. Since most such optimizations open up opportunities for further transformation, they are likely to be iterated.

The passing of annotations from the left-hand side to the right-hand side is the key here. Only those annotations that appear on the right-hand side may have analysis results fed back to them, and the location of each annotation determines the quality of the feedback. For example, in the first clause of *optz*, the expressions $\texttt{e}^{h}$ and $\texttt{arg}^{j}$ are treated atomically. As a result, there is no issue of passing their annotations forward. But the same is not true for $\texttt{(let b in e}^{h}\texttt{)}^{i}$: the annotation i belongs to a semantic entity that no longer exists after the transformation. Thus, it is dropped, and there will not be any result that concerns exactly the expression `(let b in e)`. We annotate the newly emerging entity $\texttt{(e}^{h}\ \texttt{arg}^{j}\texttt{)}$ with k because it takes the whole source expression to include both `e` and `arg`. As we can see from this example, the manipulation of indices generally requires human ingenuity, with a deep understanding of the semantics of the transformed language, something that cannot be reduced to a simple-minded consistency property. As we will see shortly, the generated putback functions, which we believe to be perfectly sensible, do not in general obey consistency .

To see how the above transformation works, we consider the following expression.

$$
\begin{array}{l}
\texttt{(let (a,b) = (let y = True}^{h} \\
\qquad\qquad\texttt{in ((y}^{i1}\texttt{+1}^{i2}\texttt{)}^{i3}\texttt{,y}^{i4}\texttt{)}^{i5}\texttt{)}^{k1} \\
\texttt{ in (a}^{j1}\texttt{ + 1}^{j2}\texttt{)}^{j3}\texttt{)}^{k2}
\end{array}
$$

Applying the second clause moves the binding `y = True` to the top level.

$$
\begin{array}{l}
\texttt{(let y = True}^{h} \\
\texttt{ in (let (a,b) = ((y}^{i1}\texttt{+1}^{i2}\texttt{)}^{i3}\texttt{,y}^{i4}\texttt{)}^{i5} \\
\qquad\texttt{in (a}^{j1}\texttt{ + 1}^{j2}\texttt{)}^{j3}\texttt{)}^{k2}\texttt{)}^{k2}
\end{array}
$$

Applying the third clause replaces the **let**s with **case**s.

$$\texttt{(case True}^{h}\texttt{ of y} \rightarrow \texttt{(case((y}^{i1}\texttt{+1}^{i2}\texttt{)}^{i3}\texttt{,y}^{i4}\texttt{)}^{i5}\texttt{ of (a,b)} \rightarrow \texttt{(a}^{j1}\texttt{ + 1}^{j2}\texttt{)}^{j3}\texttt{)}^{k2}\texttt{)}^{k2}$$

To be able to recover the original expression, we need to bidirectionalize *optz*. In this case, there are no function calls on the right-hand side, and the complement function only records which clauses are applied.

$$\begin{aligned}
&\mathit{optz}^{\bullet}\ \texttt{((let b in e}^{h}\texttt{)}^{i}\texttt{ arg}^{j}\texttt{)}^{k} &&= \mathit{Optz1}\ i\\
&\mathit{optz}^{\bullet}\ \texttt{(let x = (let b in e1}^{h}\texttt{)}^{i}\texttt{ in e2}^{j}\texttt{)}^{k} &&= \mathit{Optz2}\ i\\
&\mathit{optz}^{\bullet}\ \texttt{(let v = e1}^{h}\texttt{ in e2}^{i}\texttt{)}^{j} &&= \mathit{Optz3}
\end{aligned}$$

Tupling gives us the following.

$$\begin{aligned}
&\mathit{optz}^{\gg}\ \texttt{((let b in e}^{h}\texttt{)}^{i}\texttt{ arg}^{j}\texttt{)}^{k} &&= (\texttt{((let b in (e}^{h}\texttt{ arg}^{j}\texttt{)}^{k}\texttt{)}^{k}, \mathit{Optz1}\ i)\\
&\mathit{optz}^{\gg}\ \texttt{(let x = (let b in e1}^{h}\texttt{)}^{i}\texttt{in e2}^{j}\texttt{)}^{k} &&= (\texttt{((let b}\\
&&&\quad\texttt{in (let x = e1}^{h}\texttt{ in e2}^{j}\texttt{)}^{k}\texttt{)}^{k},\\
&&&\quad \mathit{Optz2}\ i)\\
&\mathit{optz}^{\gg}\ \texttt{(let v = e1}^{h}\texttt{ in e2}^{i}\texttt{)}^{j} &&= (\texttt{((case e1}^{h}\texttt{ of v} \rightarrow\texttt{e2}^{i}\texttt{)}^{j}, \mathit{Optz3})
\end{aligned}$$

The function *optz* is non-linear because of duplication, which requires a bit of effort in the inverting step. First of all, we separate non-linearity in expressions and annotations. For duplicated expression variables, we can safely assume that they always hold the same value, since an analysis only updates the annotations. For annotation variables, which are updated individually, we choose one value among many. For example, consider annotations as boolean values representing the presence of type errors; we have the following *put* function.

$$\begin{aligned}
&\mathit{optz}^{\ll}\ (\texttt{((let b in (e}^{h}\texttt{ arg}^{j}\texttt{)}^{k1}\texttt{)}^{k2}, \mathit{Optz1}\ i)\\
&\quad = \texttt{((let b in e}^{h}\texttt{)}^{i}\texttt{ arg}^{j}\texttt{)}^{\mathit{choose}(k1,k2)}\\
&\mathit{optz}^{\ll}\ (\texttt{ (let x = (let b in e1}^{h}\texttt{)}^{i}\texttt{in e2}^{j}\texttt{)}^{k}, \mathit{Optz2}\ i)\\
&\quad = \texttt{(let x = (let b in e1}^{h}\texttt{)}^{i}\texttt{in e2}^{j}\texttt{)}^{\mathit{choose}(k1,k2)}\\
&\mathit{optz}^{\ll}\ (\texttt{((case e1}^{h}\texttt{ of v} \rightarrow\texttt{e2}^{i}\texttt{)}^{j}, \mathit{Optz3})\\
&\quad = \texttt{(let v = e1}^{h}\texttt{ in e2}^{i}\texttt{)}^{j}
\end{aligned}$$

Here, the function choose is characterized as follows.

class $T\ a$ **where**
  $choose :: (a, a) \to a$
instance $T\ Bool$ **where**
  $choose\ (x, y) = x \parallel y$

Applying the putback transformation $optz^<$ to the type checked expression (we use T for *True* and F for *False*), we get:

```
(case True^F of y →
    (case ((y^F+1^F)^T,y^F)^F of (a,b) → (a^F + 1^F)^F)^F)^F
```

which, together with the complement, produces

```
(let (a,b) = (let y = True^F
              in ((y^F+1^F)^T,y^F)^F)^F
 in (a^F + 1^F)^F)^F
```

which maps the error to the source location that one expects. As mentioned before, this treatment of duplication breaks consistency, but is reasonable for the application here.

4.4.2.2 Removing expressions

The most straightforward form of expression removal is dead code elimination. However, this transformation is of limited interest in our application of mapping analysis results of the transformed program back to the source, since there won't be any analysis of the discarded code. Nevertheless, dead code elimination may still be part of a series of transformations, and so will need to be bidirectionalized.

A more intricate form of expression removal involves replacing an existing expression with a new one. Common examples include beta reduction and variable substitution; beta reduction directly reduces function applications, while substitution of variables is one of the most effective ways of defacing a program. Since the source expressions are replaced instead of discarded, it is necessary to map the analysis results of the replacement back to the original. We demonstrate by adding clauses to the optimizing transformation seen earlier.

$optz$ `(case (C e1`^{h1} `...  en`^{hn}`)`^{i} `of C x1 ...  xn →  e`^{j}`; alts)`$^k = e'$
  where $e' = subst\ ([$`(e1`^{h1}`,`$x1$`)`$, ..., $`(en`^{hn}`,`$xn$`)`$], $\mathtt{e}^j)$

$$
\begin{aligned}
&\mathit{optz}\ \ \texttt{((}\lambda\ \texttt{x} \rightarrow \texttt{e}^{h}\texttt{)}^{i}\ \texttt{arg}^{j}\texttt{)}^{k} && = e' \\
&\quad \textbf{where}\ e' = \mathit{subst}\ (\texttt{(arg}^{j}\texttt{,}x\texttt{)}, \texttt{e}^{h})
\end{aligned}
$$

The first clause above eliminates a **case** scrutiny when the constructor is known and removes the dead alternatives, while the second clause performs a step of beta reduction. Both cases rely on the substitution of variables, which is defined below.

$$
\begin{aligned}
&\mathit{subst}\ ([\,], \texttt{x}) && = \texttt{x} \\
&\mathit{subst}\ ((\texttt{e}, v) : ss, \texttt{x}) \mid \texttt{x} \equiv \texttt{v} && = \texttt{e} \\
&\mathit{subst}\ ((\texttt{e}, v) : ss, \texttt{x}) && = e' \\
&\quad \textbf{where}\ e' = \mathit{subst}\ (ss, \texttt{x})
\end{aligned}
$$

The function *subst* takes a substitution (a list of key-value pairs) and replaces variables in an expression that appear as keys. We use the notation `v` to denote the type variable constructed from the name v. Note that we omit the boilerplate traversal of other syntactic constructs of expressions and only highlight the case of variables.

$$
\begin{aligned}
&\mathit{optz}^{\bullet}\ \ \texttt{(case (C e1}^{h1}\ \texttt{...}\ \ \texttt{en}^{hn}\texttt{)}^{i}\ \texttt{of C x1 ...}\ \ \texttt{xn} \rightarrow \texttt{e}^{j}\texttt{; alts)}^{k} \\
&\quad = \mathit{Optz4}\ (\mathit{subst}^{\bullet}\ ([\texttt{(e1}^{h1}\texttt{,}x1\texttt{)}, ..., \texttt{(en}^{hn}\texttt{,}xn\texttt{)}], \texttt{e}^{j}))\ \ \texttt{alts}\ \ i\ k \\
&\quad \textbf{where}\ e' = \mathit{subst}\ ([\texttt{(e1}^{h1}\texttt{,}x1\texttt{)}, ..., \texttt{(en}^{hn}\texttt{,}xn\texttt{)}], \texttt{e}^{j}) \\
&\mathit{optz}^{\bullet}\ \ \texttt{((}\lambda\ \texttt{x} \rightarrow \texttt{e}^{h}\texttt{)}^{i}\ \texttt{arg}^{j}\texttt{)}^{k} \\
&\quad = \mathit{Optz5}\ (\mathit{subst}^{\bullet}\ (\texttt{(arg}^{j}\texttt{,}x\texttt{)}, \texttt{e}^{h}))\ i\ k \\
&\quad \textbf{where}\ e' = \mathit{subst}\ (\texttt{(arg}^{j}\texttt{,}x\texttt{)}, \texttt{e}^{h})
\end{aligned}
$$

The complement function for *optz* records the eliminated dead code, whereas the key to bidirectionalization lies in the undoing of substitutions.

$$
\begin{aligned}
&\mathit{subst}^{\bullet}\ ([\,], \texttt{x}) && = \mathit{Subst1} \\
&\mathit{subst}^{\bullet}\ ((\texttt{e}, v) : ss, \texttt{x}) \mid \texttt{x} \equiv \texttt{v} && = \mathit{Subst2}\ v\ ss\ \ \texttt{x} \\
&\mathit{subst}^{\bullet}\ ((\texttt{e}, v) : ss, \texttt{x}) && = \mathit{Subst3}\ (\mathit{subst}^{\bullet}\ (ss, \texttt{x}))\ \ \texttt{e}\ \ v \\
&\quad \textbf{where}\ e' = \mathit{subst}\ (ss, \texttt{x})
\end{aligned}
$$

Tupling gives us the following.

$$
\begin{aligned}
&\mathit{optz}^{\gg}\ \ \texttt{(case (C e1}^{h1}\ \texttt{...}\ \ \texttt{en}^{hn}\texttt{)}^{i}\ \texttt{of C x1 ...}\ \ \texttt{xn} \rightarrow \texttt{e}^{j}\texttt{; alts)}^{k} \\
&\quad = (e', \mathit{Optz4}\ c\ \ \texttt{alts}\ \ i\ k) \\
&\quad\quad \textbf{where}\ (e', c) = \mathit{subst}^{\gg}\ ([\texttt{(e1}^{h1}\texttt{,}x1\texttt{)}, ..., \texttt{(en}^{hn}\texttt{,}xn\texttt{)}], \texttt{e}^{j}) \\
&\mathit{optz}^{\gg}\ \ \texttt{((}\lambda\ \texttt{x} \rightarrow \texttt{e}^{h}\texttt{)}^{i}\ \texttt{arg}^{j}\texttt{)}^{k}
\end{aligned}
$$

$$
\begin{aligned}
&= (e', \mathit{Optz5}\ c\ i\ k) \\
&\quad \textbf{where}\ (e', c) = \mathit{subst}^{\gg}\ ((\mathtt{arg}^{j}, x), \mathtt{e}^{h})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subst}^{\gg}\ ([\,], \mathtt{x}) && = (\mathtt{x}, \mathit{Subst1}) \\
&\mathit{subst}^{\gg}\ ((\mathtt{e}, v) : ss, \mathtt{x}) \mid \mathtt{x} \equiv \mathtt{v} && = (\mathtt{e}, \mathit{Subst2}\ v\ ss\ \ \mathtt{x}) \\
&\mathit{subst}^{\gg}\ ((\mathtt{e}, v) : ss, \mathtt{x}) && = (e', \mathit{Subst3}\ c\ \ \mathtt{e}\ \ v) \\
&\quad \textbf{where}\ (e', c) = \mathit{subst}^{\gg}\ (ss, \mathtt{x})
\end{aligned}
$$

Since both functions in this section are linear, the inverting step is therefore straightforward. Consider a variant of the result of previous transformations in the last section.

$$
\begin{aligned}
&\mathtt{(case\ True}^{F}\ \mathtt{of\ y} \rightarrow \\
&\quad \mathtt{(case\ (y}^{F}\mathtt{,y}^{F}\mathtt{)}^{F}\ \mathtt{of\ (a,b)} \rightarrow \mathtt{(a}^{F}\ \mathtt{+\ 1}^{F}\mathtt{)}^{F}\mathtt{)}^{F}\mathtt{)}^{F}
\end{aligned}
$$

A further step of transformation eliminates the scrutiny of the known constructor, and type checking the translated code reports an error.

$$
\mathtt{(case\ True}^{F}\ \mathtt{of\ y} \rightarrow \mathtt{(y}^{F}\ \mathtt{+\ 1}^{F}\mathtt{)}^{T}\mathtt{)}^{F}
$$

Despite the drastic change in appearance, the putback function easily recovers the original expression with the correct location of the error.

$$
\begin{aligned}
&\mathtt{(case\ True}^{F}\ \mathtt{of\ y} \rightarrow \\
&\quad \mathtt{(case\ (y}^{F}\mathtt{,y}^{F}\mathtt{)}^{F}\ \mathtt{of\ (a,b)} \rightarrow \mathtt{(a}^{F}\ \mathtt{+\ 1}^{F}\mathtt{)}^{T}\mathtt{)}^{F}\mathtt{)}^{F}
\end{aligned}
$$

4.4.2.3 Creating expressions

Program transformations very often turn programs into more verbose forms with simpler constructs. A good example of this is desugaring. Consider the translation of list comprehensions in Haskell seen in Section 4.1.

$$
\begin{aligned}
&\mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ True}^{i}\ \mathtt{]}^{j} = \mathtt{[e}^{h}\mathtt{]}^{j} \\
&\mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ q}^{i}\ \mathtt{]}^{j} \quad = \mathit{lst} \\
&\quad \textbf{where}\ \mathit{lst} = \mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ q}^{i}\mathtt{,True}^{j}\ \mathtt{]}^{j} \\
&\mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ b}^{i}\mathtt{,Q\ ]}^{j} \ = \ \mathtt{(if\ b}^{i}\ \mathtt{then}\ \mathit{lst}\ \mathtt{else\ []}^{j}\mathtt{)}^{j} \\
&\quad \textbf{where}\ \mathit{lst} = \mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ Q\ ]}^{j} \\
&\mathit{compr}\ \ \mathtt{[\ e}^{h}\ \mathtt{|\ p\leftarrow l}^{i}\mathtt{,Q\ ]}^{j} = \ \mathtt{(let\ f\ x\ =\ case\ x\ of\ p}\ \rightarrow\ \mathit{lst} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad _\ \ \rightarrow\ \mathtt{[]}^{j}
\end{aligned}
$$

$$\texttt{in ((concatMap}^{j}\ \texttt{f}^{j}\texttt{)}^{j}\ \texttt{l}^{i}\texttt{)}^{j}\texttt{)}^{j}$$

where $lst = compr\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j}$

The translated code uses semantically simpler constructs and is necessarily more verbose. For example, in the last case above, a generator is turned into a function that takes care of the pattern matching and an invocation of the higher-order function *concatMap* for the generation. This kind of creation is generally problematic for analysis result reporting, since the expressions involved may not exist in the source, and causes considerable confusion. And this is the reason for the wild-spread of annotation duplication in the above program, which signals the absence of clear connections between certain input and output expressions. The complement function for *compr* is the following.

$$\begin{aligned}
&compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| True}^{i}\ \texttt{]}^{j} &&= Compr1\ i\\
&compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\ \texttt{]}^{j} &&= Compr2\ (compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j})\\
&\quad \textbf{where}\ lst = compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j}\\
&compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j} &&= Compr3\ (compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j})\\
&\quad \textbf{where}\ lst = compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j}\\
&compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| p←l}^{i}\texttt{,Q]}^{j} &&= Compr4\ (compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j})\\
&\quad \textbf{where}\ lst = compr^{\bullet}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j}
\end{aligned}$$

The tupling step is standard.

$$\begin{aligned}
&compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| True}^{i}\ \texttt{]}^{j} &&= (\texttt{[e}^{h}\ \texttt{]}^{j}, Compr1\ i)\\
&compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\ \texttt{]}^{j} &&= (lst, Compr2\ c)\\
&\quad \textbf{where}\ (lst, c) = compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j}\\
&compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j} &&= (\texttt{((if b}^{i}\ \texttt{then}\ lst\ \texttt{else []}^{j}\texttt{)}^{j}, Compr3\ c)\\
&\quad \textbf{where}\ (lst, c) = compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j}\\
&compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| p←l}^{i}\texttt{,Q]}^{j} &&= (\texttt{(let f x = case x of p} \rightarrow lst\\
&&&\qquad\qquad\qquad\qquad \texttt{_} \rightarrow \texttt{[]}^{j}\\
&&&\quad \texttt{in ((concatMap}^{j}\ \texttt{f}^{j}\texttt{)}^{j}\ \texttt{l}^{i}\texttt{)}^{j}\texttt{)}^{j}, Compr4\ c)\\
&\quad \textbf{where}\ (lst, c) = compr^{\gg}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{j}
\end{aligned}$$

We deal with the duplications by ignoring the ones that do not have clear source origins.

$$\begin{aligned}
&compr^{<}\ (\texttt{[e}^{h}\ \texttt{]}^{j}, Compr1\ i) &&= \texttt{[e}^{h}\ \texttt{| True}^{i}\ \texttt{]}^{j}\\
&compr^{<}\ (lst, Compr2\ c) &&= \texttt{[e}^{h}\ \texttt{| q}^{i}\ \texttt{]}^{j}\\
&\quad \textbf{where}\ \ \texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{-}\ \texttt{]}^{j} = compr^{<}\ (lst, c)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{compr}^{<}\ (\ \texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j}, \mathit{Compr3}\ c) && = \ \texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j}\\
&\quad \textbf{where}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{-} = \mathit{compr}^{<}\ (\mathit{lst}, c)\\
&\mathit{compr}^{<}\ (\texttt{(let f x = case x of p} \rightarrow \mathit{lst}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{_} \rightarrow \texttt{[]}^{-}\\
&\qquad\qquad \texttt{in ((concatMap}^{-}\ \texttt{f}^{-})^{-}\ \texttt{l}^{i})^{-})^{j}, \mathit{Compr4}\ c) = \ \texttt{[e}^{h}\ \texttt{| p}\leftarrow\texttt{l}^{i}\texttt{,Q]}^{j}\\
&\quad \textbf{where}\ \ \texttt{[e}^{h}\ \texttt{| Q]}^{-} = \mathit{compr}^{<}\ (\mathit{lst}, c)
\end{aligned}
$$

This treatment of duplication means whatever happens to those ignored annotations will not be reflected in the source. However, we argue that since the transformations are assumed correct, whatever type errors must arise from the source, and are unlikely to show up exclusively in the ignored annotations, which represent expressions that do not have clear source connections. For example, consider the ill typed *quicksort* given in the beginning of this chapter. Though the error will involve the generated *concatMap* and f whose annotations are ignored by the putback function, it is sufficient to reflect the annotation h back to the source to get a good hint of the source of the error.

4.4.2.4 Fresh Variables

Up to now, we have only mentioned in passing that GHC avoids name capture by always inventing fresh variables. As a matter of fact, the newly created function `f` above is expected to have a fresh name. The primitive way of achieving this in pure functional languages (circumventing the need for a global mutable state) is to pass a name generator as an argument.

$$
\begin{aligned}
&\mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| True}^{i}\ \texttt{]}^{j}, n) = (\texttt{[e}^{h}\texttt{]}^{j}, n)\\
&\mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| q}^{i}\ \texttt{]}^{j}, n) = (\mathit{lst}, n')\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j}, n)\\
&\mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j}, n) = (\texttt{(if b}^{i}\ \texttt{then}\ \mathit{lst}\ \texttt{else []}^{j})^{j}, n')\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| Q]}j, n)\\
&\mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| p}\leftarrow\texttt{l}^{i}\texttt{,Q]}^{j}, n) = (\texttt{(let n x = case x of p} \rightarrow \texttt{lst}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{_} \rightarrow \texttt{[]}^{j}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in ((concatMap}^{j}\ \texttt{f}^{j})^{j}\ \texttt{l}^{i})^{j})^{j}, n')\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| Q]}^{j}, \mathit{new}\ n)
\end{aligned}
$$

In this version, the name `n` is taken from the generator n, and a new generator is used in the recursive call. A pleasant observation is that adding name generators to program

transformations does not complicate the bidirectionalization process, as shown in the following complement function.

$$
\begin{aligned}
&\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| True}^{i}\ \texttt{]}^{j}, n) = \mathit{Compr1}\ i\\
&\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| q}^{i}\ \texttt{]}^{j}, n) = \mathit{Compr2}\ (\mathit{comprc1c}\ (\texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j}, n))\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| q}^{i}\texttt{,True}^{j}\ \texttt{]}^{j}, n)\\
&\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| b}^{i}\texttt{,Q]}^{j}, n) = \mathit{Compr3}\ (\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| Q]}^{j}, n))\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| Q]}^{j}, n)\\
&\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| p←l}^{i}\texttt{,Q]}^{j}, n) = \mathit{Compr4}\ (\mathit{compr1}^{\bullet}\ (\texttt{[e}^{h}\ \texttt{| Q]}^{j}, n))\\
&\quad \textbf{where}\ (\mathit{lst}, n') = \mathit{compr1}\ (\texttt{[e}^{h}\ \texttt{| Q]}^{j}, \mathit{new}\ n)
\end{aligned}
$$

This code is almost identical to the previous version. The name generators do not influence the structural traversal, and they are always used on the right-hand side. Thus, adding them has no impact on the construction of complements. Since the generator is not subject to updating in the view, the tupling and inverting steps are standard.

4.5 Discussion

As we have seen, the use of bidirectional programming for analysis result reporting is a promising technique. Program analysis can now be performed at any stage of a transformation process, and the generated putback function is able to map the results back to the source code. Our examples have demonstrated some of the most interesting kinds of program transformations that happen in compilers. We are yet to conduct a survey on DSLs. Nevertheless, we expect many cases to be less intricate than what we have seen. For example, in a related work of handling unit tests specified in a DSL but executed in a general purpose language (GPL) [WGM09], it is shown that even with a very simple one-to-many assumption, where a line of DSL code always translates to multiple lines of GPL code, considerable coverage can be achieved.

There has already been a body of work on improving type error message reporting or even diagnosis [HHS03, HLvI03, SSW04]. Our proposal is not in competition, but complements these. We do not study how existing type checking/inference algorithms can be improved. Instead, we rely on the given results and try to map them to the source. In particular, the system in [SSW04] is centred around the identification of sets of program locations contributing to a particular error, and marks the locations by annotating the AST; our proposal in this chapter applies directly. For example, consider the previously seen example of reducing the expression

```
(let (a,b) = (let y = True
              in (y + 1,y))
 in (a + 1))
```

to

```
(case True of y → y + 1)
```

Applying the technique in [SSW04] to the reduced expression above produces

$$\texttt{(case True}^{T}\texttt{ of y} \rightarrow \texttt{(y}^{F}\texttt{ + 1}^{F}\texttt{)}^{T}\texttt{)}^{F}$$

where there are two locations marked as a minimal set that contributes to the error. Running the program transformation backwards gives us the following.

$$\begin{array}{l}\texttt{(let (a,b) = (let y = True}^{T} \\ \qquad\qquad\texttt{in ((y}^{F}\texttt{+1}^{F}\texttt{)}^{T}\texttt{,y}^{F}\texttt{)}^{F}\texttt{)}^{F} \\ \texttt{ in (a}^{F}\texttt{ + 1}^{F}\texttt{)}^{F}\texttt{)}^{F}\end{array}$$

which is exactly the result we will get by applying the same diagnosis technique directly to it.

4.5.1 Reuse of Existing Code

Our proposal comes with a small price: the transformations must operate on annotated ASTs, and so are better suited for fresh developments.

In the case where an unannotated AST and its transformation already exist, it will be highly desirable to reuse them. The folklore way, consolidated in [vSMJ10], of increasing genericity of datatypes is through the so-called *open recursion* form.

$$\begin{array}{l}\textbf{data}\ \mathit{ExpF}\ r = \mathit{Var}\ \mathit{String} \\ \qquad\qquad\quad \mid \mathit{INT}\ \mathit{Int} \\ \qquad\qquad\quad \mid \mathit{Lam}\ \mathit{String}\ r \\ \qquad\qquad\quad \mid \mathit{App}\ r\ r\end{array}$$

The above definition is often known as the *base functor* of the datatype *Exp* (see Section 4.4.1), where a type argument is abstracted and awaits filling in. As a result, the shape of the datatype is fixed and can be reused for different instantiations. For example, the original *Exp* datatype can be constructed through the following fixed-point.

newtype $Exp = Ex\ (ExpF\ Exp)$

Annotations can be added at every level of recursion in the type now.

newtype $ExpA = ExA\ (Ann, ExpF\ ExpA)$

The job of adding annotations to the expression type is now a one-liner. However, this approach does require foresight of the programmer to design for adaptation from the very beginning. Moreover, the mandatory constructors introduced by using **newtype** in Haskell, such as *Ex* and *ExA*, are inconvenient to work with. The reuse of function definitions is restricted to the recursion pattern, say regular structural recursion. This is not suitable for our application: most transformations' recursion patterns are not regular, and even when they are, it is unlikely to be made explicit with a higher order function *fold* to enable the technique above.

This discussion is also related to the refactoring framework developed in Chapter 3, where a program manipulating bare ASTs could be refactored into one handling annotated ASTs. Nevertheless, the programmer must re-implement the functions that performs annotation propagations, such as the program transformations we have seen in this chapter, but may hope that other functions not concerning the annotations are spared from the change. If we see the bare ASTs and annotated ASTs as the abstract and concrete representations respectively, the abstraction is a fold with a projection function that strips the annotations at all levels; and its right-inverse annotates the structure with default values. Strictly speaking, this projection function that removes the annotations cannot be implemented in RINV, because annotations are not "redundant" information that can be derived from the ASTs. A pragmatic way to get around this problem is to make sure that any annotation setting operations, such as type checking in our application, are invoked towards the end of an execution and finish without converting back to the abstract representation.

4.5.2 Syntactic vs. Semantic Bidirectionalization

In the previous sections, we have spent much effort extending the original syntactic bidirectionalization technique and studying its updatability property. The investigation is certainly interesting on its own, but given that the semantic approach provides sufficient updatability for our purpose too, there is an alternative.

Generally speaking, the semantic approach [Voi09, Section 2.1.2.2] has the advantage of not requiring the source code of the transformations, but at the cost of run-time

performance: the dictionary look-ups are not expected to be cheap. Assuming linear time complexity for retrieval functions, the semantic approach produces putback functions with complexity $O\ (n\ \times\ lg\ n)$, which is significantly worse than the linear time putback function generated by our approach. In our application, since transformations (retrieval functions) are expected to be implemented freshly based on the annotated expressions, the benefit of semantic bidirectionalization does not really apply.

In either case, our use of bidirectional programming suggests a wider applicability of the techniques, other than the classic view-updates of databases. As a matter of fact, many of the datatype-based language designs [MHT04b, MHT04a, HMT04, PC10], including the two that have been mentioned above [MHN$^+$07, Voi09], have already shown the potential of being more general-purpose. But study of their applications appears to lag behind. We believe the results in this chapter may help to shape the future direction of bidirectional language design.

Chapter 5

Looking from the Top

5.1 Introduction

We have seen two examples in the previous chapters where the update requirements play a central role in the design of bidirectional languages. In particular, for program analysis result reporting in Chapter 4, a view, being the annotated translated code, is only changed in a very specific way, whereas for refactoring pattern matching in Chapter 3, programming with abstract representations allows general updates that potentially change the type of a view. Both of the above are specific problems and demand specific solutions. In this chapter, we take a different perspective by looking at how update information can be used to improve bidirectional programming in general.

So far, we have used the word "update" to describe both changes in views and those in sources. In this chapter, we restrict the use of *update* to the transformed effect on a source as a result of an *edit* to the view. Despite being fundamental to bidirectional programming, there is no universal agreement on what constitutes an edit. Roughly speaking, opinions are divided as to whether one should look into the mechanism of an edit or simply its result. Translated into language design, one can either take an operation-based approach considering an editing function that changes a view, or a state-based approach that only sees the unedited and edited views. It happens that the majority of existing bidirectional systems take the latter, due to its mathematical simplicity and good compatibility. The bidirectional laws we have discussed at the beginning of the thesis are specified in a state-based system. Since only the updated view is required for the putback function, it is easier to design a bidirectional system independently from any editing system. On the other hand, a state-based approach necessarily discards information about an edit, and only tries to reverse-engineer it later by performing a kind of difference analysis on the two view values. Consequently, the run-time performance of state-based backward functions is bound by the linear barrier: even a small change to the view implies a complete re-traversal.

Meertens [Mee98, Section 2.1.1.4] observed that to maintain constraints between two structures, it is useful to know how a view is edited. Consider the scenario that two lists are connected by a mapping relation (i.e. one is the result of applying function *map* to the other, and vice versa); an edit to one list, say deletion at position indexed by n, can be translated to a deletion at the corresponding position in the other list. In this setting, a lot more information about the editing is made available to the bidirectional system, including where (the position index) and what (deletion) has changed. As a result, the updating process could be more straightforward compared to a state-based approach,

where we only know that one list is changed into another list that is one element short, which is fairly ambiguous. In addition, having an operation for source update potentially achieves better-than-linear runtime performance. This is particularly attractive given that "*updates typically change a small part of the document and leave most of the data fixed*" [Che08] and "*the time to process an edit operation should be proportional to the size of the change, and (ideally) independent of the total size of the document*" [Ber09].

If run-time performance is the only concern, the "where" part of the knowledge of an update is the key; once the update-affected fragment is picked out, a state-based approach could perform the changes as efficient as an operation-based counterpart, without the undesired complications that the latter brings.

In this chapter, we propose a novel *change-based* framework for bidirectional programming. Instead of inventing from first principles to add into the already flourishing group of bidirectional systems, we focus on the preservation and propagation of user-provided editing information. In a sense, our proposal can be seen as a generic optimiser of some given state-based bidirectional systems: we exploit any locality in the editing of views, and try to translate it into incrementality in the updating of sources. Such preservation of locality obviously does not hold for arbitrary transformations. Identifying the semantic properties required forms one of the major technical contributions of this chapter. The ultimate goal of our system is to reduce an update of a (large) structure into one of a (small) delta, and then outsource the hopefully much smaller problem to existing state-based systems. This step positively impacts the run-time behaviours of putback functions as well as the quality of updates, both due to the newly gained incrementality: less "damage" to the source implies less processing and better results.

Our proposal aims at modularity: there is a clearly defined interface that decouples any editing system from our framework; and different state-based bidirectional systems can be plugged in as black-boxes, whatever their manifestation as purpose-built bidirectional languages or syntactic/semantic transformations of unidirectional program. As a result, a change-based bidirectional system arises automatically from a given state-based one, while preserving the bidirectional properties of the latter.

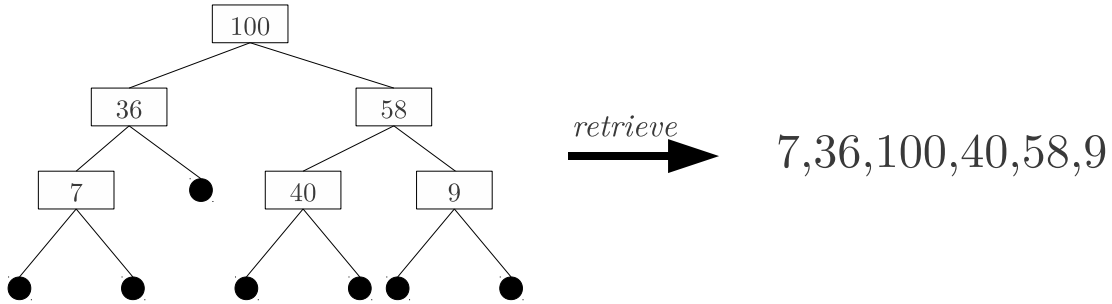
5.1.1 A Small Example

Suppose we have some source data as a binary tree:

```
data Tree = Tip
          | Node Int Tree Tree
```

$$aTree = Node\ 100\ (Node\ 36\ (Node\ 7\ Tip\ Tip)\ Tip)$$
$$(Node\ 58\ (Node\ 40\ Tip\ Tip)\ (Node\ 9\ Tip\ Tip))$$

We may want to view it as a list. It can be done with a retrieval function that performs an in-order traversal to produce a list view.



Now, suppose the number 40 is deleted from the view. A state-based putback function will take the edited view $[7, 36, 100, 58, 9]$ and try to construct a tree without the deleted element; and hopefully the new source remains similar to the original one so that unnecessary changes are kept to a minimum. Note that we have deliberately kept all the functions abstract because our proposal is not dependent on any particular implementation.

The method described above takes effort proportional to the size of the data, not to the size of the change. Assuming a functional cons-list, the deletion of 40 involves traversing the view list to the location of the deletion, and changes only the sublist $[40, 58, 9]$ rooted at that location; a more efficient approach is to update only the source subtree $(Node\ 58\ (Node\ 40\ Tip\ Tip)\ (Node\ 9\ Tip\ Tip))$, which is responsible for generating the view fragment $[40, 58, 9]$. That is to say, the bidirectional updating should be incremental. Better still, in the case of lists, where a deletion is local and does not induce subsequent changes to a substructure, a more refined analysis may discover that only the subtree $(Node\ 40\ Tip\ Tip)$ is really affected by the edit, and updating it is sufficient.

In the sequel of the chapter, we will discuss in detail how retrieval functions that support incremental updates can be identified; and a constructive method for performing the said update in a change-based framework. Assuming a fairly balanced source tree, the complexity of our system is $O\ (m\ \times\ log\ n + f\ m)$ where n is the size of the source tree, m is the size of the affected source part, and f is the complexity function of a state-based putback function. The update itself takes time proportional to m; but it takes $m\ \times\ log\ n$ time to locate the target source location.

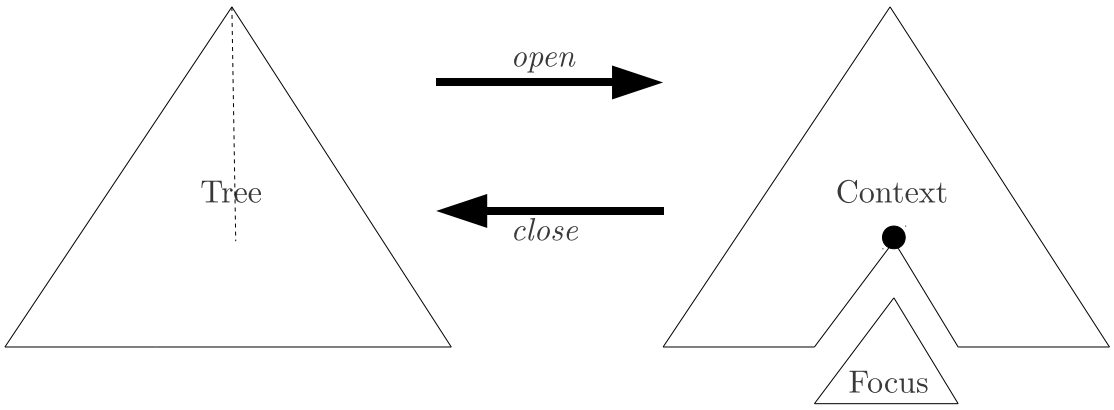


Figure 5.1: The Context-Focus Representation

5.2 The Overall Setting

We consider typed tree-structured data represented as polynomial datatypes. We assume all trees are labeled by uniquely identified elements; while the elements are editable, their identifiers are immutable and non-inventible by retrieval transformations. In this chapter, the element values in labels are not important, so we will simply say labels when we mean the unique identifiers of them.

5.2.1 Tree Navigation

To gain access to a subtree of interest, we navigate from the root following a given path; the path is usually incomplete, in a sense that it runs out before reaching a terminal leaf. When this happens, the tree that is being travelled is left in a state with two structures separated by the point where the path finishes: a lower one, being a subtree, is the current *focus*; and an upper one is the *context* of the focus (see Figure 5.1). For well-typedness, we only consider valid paths that lead to recursive components. As an example, for the binary tree datatype above, the focus can only descend to either the left or the right subtree, but not the label.

We represent a context as a tree with a hole in it, denoting the location of the subtree that is separated from it. For binary trees, contexts can be defined as the following:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ctx} = \mathit{Hole} \\
&\qquad\qquad \mid\ \mathit{LH}\ \mathit{Int}\ \mathit{Ctx}\ \mathit{Tree} \\
&\qquad\qquad \mid\ \mathit{RH}\ \mathit{Int}\ \mathit{Tree}\ \mathit{Ctx}
\end{aligned}
$$

It is not hard to see that the context type follows directly from the structure of the tree. As a matter of fact, contexts are type-indexed data types [HJL04] and can be defined generically [HJ01, McB01].

In general, for any source type s, we can select its subtrees by their locations:

$$child :: s \to Int \to s$$

and a function *zoom* opens a tree by focusing on a subtree deep inside the source following a path:

$$zoom :: s \to [Int] \to s$$
$$zoom = foldl\ child$$

Subtrees of a tree form a subterm ordering; and we are only interested in non-empty trees (i.e., trees with labels). We use a function $label :: t \to Labels$, where t is the type of the tree and *Labels* is the type of label sets, to collect all the unique label identifiers in a tree; the identifiers could be the elements themselves or some additional indices. Very often, we use a short hand $\langle x \rangle$ to denote $label\ x$.

Definition 5.1 (Labeled-Subterm Ordering) *Given trees x and y such that $\langle x \rangle \neq \emptyset$ and $\langle y \rangle \neq \emptyset$, we say x is a strict subterm of y, written as $x \prec y$ if*

$$\exists i.x = zoom\ i\ y \wedge \langle x \rangle \subset \langle y \rangle$$

We say x is a subterm of y, written as $x \preccurlyeq y$, if $x \prec y \vee x = y$. We say a subterm r is trivial if $\langle r \rangle = \emptyset$. □

Throughout this paper, we assume non-trivial subterms unless otherwise stated. Note that in addition to structural subterm ordering specified by tree opening, we require a smaller tree to always contain less labels. In another word, we do not consider trees extended with "junk" structures that are not labeled. As a remark, the situation of having a bigger tree without supplying new labels will not occur with our representation of binary trees, but has to be dealt with in general polynomial datatypes.

Fact 5.2 (Distinct Subterms) *Because tree nodes are uniquely labelled, when $r \preccurlyeq t$ (and r is non-trivial), then r is in fact at the end of a unique path; that is, there is a (partial) function $location :: t\ a \to t\ a \to Path$ satisfying*

$$location\ t\ r = p \Leftrightarrow r = zoom\ t\ p$$

We say that "r is at depth n in t" when $n = length\ (location\ t\ r)$. □

Definition 5.3 (Orderedness) *We say* x *and* y *are ordered, written as* $x \sim y$*, if* $x \preccurlyeq y \vee y \preccurlyeq x$. □

Note that the above definition shall not be confused with a equivalence relation where a conjunction, instead of a disjunction, is used in the premise. Given orderedness, the subterm relation is equivalent to the subset relation among label sets.

Fact 5.4 $x \preccurlyeq y \Leftrightarrow (\langle x \rangle \subseteq \langle y \rangle \wedge x \sim y)$

□

And the label sets of two ordered trees overlap.

Fact 5.5 $x \sim y \Rightarrow \langle x \rangle \cap \langle y \rangle \neq \emptyset$ □

We use two infix operators to obtain the context that is left out by a tree opening, and close a context with a focus to recover the source tree. The two operators are of the same precedence and associate to the right.

$$(/) :: s \to s \to Ctx\ s$$
$$(\lessdot) :: Ctx\ s \to s \to s$$

The closing function $(\lessdot)$ attaches a tree to the hole in a context. For binary trees, it can be defined as the following:

$$(\lessdot) :: Ctx \to Tree \to Tree$$
$$Hole \lessdot t = t$$
$$LH\ a\ c\ rt \lessdot t = Node\ a\ (c \lessdot t)\ rt$$
$$RH\ a\ lt\ c \lessdot t = Node\ a\ lt\ (c \lessdot t)$$

The result of a closing extends the input tree, which is captured by a monotonicity requirement.

Requirement 5.6 (Monotonicity) $y \preccurlyeq x \lessdot y$ □

Interested readers may refer to the discussion of one-hole context in [McB01] for a more formal treatment of the closing function.

The subtracting function $(/)$ forms a Galois connection with $(\lessdot)$, with equality as the orderings:

Requirement 5.7 (Galois Connection) *Given that* $x \preccurlyeq y$*, then* $c \lessdot x = y \Leftrightarrow y/x = c$

□

and can be nested

Requirement 5.8 (Nesting) *Given that* $x \preccurlyeq y \preccurlyeq z$*, then*

$$(z \mathbin{/} y) \lessdot (y \mathbin{/} x) \lessdot w = (z \mathbin{/} x) \lessdot w$$

□

From the Galois Connection, we can easily derive that $(/)$ and $(\lessdot)$ cancels each other.

Fact 5.9 (Cancellation) *Given that* $y \preccurlyeq x$*, then* $(x \mathbin{/} y) \lessdot y = x$ *and* $(c \lessdot x) \mathbin{/} x = c$*.*

□

The above requirements completely specifies the subtracting function $(/)$; for binary trees, it can be defined as the following:

$$\begin{aligned}
&(/) :: \mathit{Tree} \to \mathit{Tree} \to \mathit{Ctx} \\
&t \mathbin{/} t' = \mathbf{case}\ \mathit{dig}\ t\ (\mathit{root}\ t')\ \mathit{of}\ \mathit{Just}\ c \to c
\end{aligned}$$

$$\begin{aligned}
&\mathit{dig} :: \mathit{Tree} \to \mathit{Int} \to \mathit{Maybe}\ \mathit{Ctx} \\
&\mathit{dig}\ (\mathit{Node}\ a\ \mathit{lt}\ \mathit{rt})\ b \mid a \equiv b \qquad = \mathit{Just}\ \mathit{Hole} \\
&\qquad\qquad\qquad\qquad \mid \mathit{otherwise} = \mathbf{case}\ (l, r)\ \mathit{of}\ (\mathit{Just}\ c, _) \to \mathit{Just}\ (\mathit{LH}\ a\ c\ \mathit{rt}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (_, \mathit{Just}\ c) \to \mathit{Just}\ (\mathit{RH}\ a\ \mathit{lt}\ c) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (_, _) \qquad\ \to \mathit{Nothing} \\
&\quad \mathbf{where}\ l = \mathit{dig}\ \mathit{lt}\ b \\
&\qquad\quad\ r = \mathit{dig}\ \mathit{rt}\ b \\
&\mathit{dig}\ \mathit{Tip}\ _ \qquad\qquad\qquad\qquad = \mathit{Nothing}
\end{aligned}$$

5.2.2 Local Editing

An *editing function* is a function with the same input and output types $\mathit{edit} :: v \to v$. We require all editing functions to be total so that they can always be applied to a subterm of a view. Locality of an editing function is in some sense context-independent, where an editing function can be promoted up through a tree outside the subterm that is affected by the edit.

Definition 5.10 (Locality) *We say an editing function e on a view v is local to a subterm $u0$ of v if*

$$\forall u.u0 \preccurlyeq u \preccurlyeq v \Rightarrow e\ v = (v\ /\ u) \lessdot e\ u$$

□

In the above definition, applying the local editing function e to any subterm u of v no less than the affected subterm $u0$ has the same effect as applying the function to v. For example, as we have seen in the binary tree example, deleting 40 from the sublist $[40, 58, 9]$ and combining the result with the context $[7, 36, 100]$ is the same as deleting 40 from the complete view $[7, 36, 100, 40, 58, 9]$ (and then combining the result with the trivial context $[\,]$). There is certainly an ordering among different levels of locality, based on the ordering of $u0$, which falls out from the above definition. In this sense, a context-sensitive (path-based) editing function, such as deleting the root of the input tree, is never good, as the $u0$ will have to be v. We will discuss the option for remedying this in Section 5.6.1.

In our proposal, the subterm $u0$ that an editing function is local to is user-provided; and our technique is based on the assumption that $u0$ is significantly smaller than v. We pair the editing function with an additional function that returns the affected subterm.

$$\textbf{data}\ \mathit{Edit}\ v = E\ \{\mathit{edit} :: v \to v, \mathit{affect} :: v \to v\}$$

such that for a given v, *edit* is local to *affect* v.

5.2.3 Changed-based Bidirectional Systems

A change-based bidirectional system consists two total functions: a retrieval function $f :: s \to v$ from source to view, and a putback function $f^{<}_{ch} :: \mathit{Edit}\ v \to s \to s$. We only consider retrieval functions that are regular structural recursions, because they are more likely to benefit from our proposed improvement. We will discuss this choice in detail in Section 5.3.2. We also rule out retrieval functions involving duplication of data, so that labels' unique identifications are preserved by the retrieval transformation. The putback function $f^{<}_{ch}$ is higher-order, in contrast to $f^{<}_{st} :: (v, s) \to s$ in a state-based setting. Thus $f^{<}_{ch}$ no longer constructs an updated source from an edited view, but from the original source; any information in the edited view can be derived from the editing function and the original source. In contrast to an operation-based approach, $f^{<}_{ch}$ is not dependent on the actual editing functions.

Bidirectional laws semantically equivalent to those developed for state-based bidirectional systems can be specified in the new setting.

Consistency $f\ (f^{<}_{ch}\ edt\ s) = edit\ edt\ (f\ s)$

Acceptability $f^{<}_{ch}\ (E\ \{\, edit = id \,\}) = id$

Undoability $f^{<}_{ch}\ (edt\ \{\, edit = (edit\ edt)^{\circ} \,\}) \circ f^{<}_{ch}\ edt = id.$

Moving from a state-based system to a change-based system potentially improves runtime performance as we exploit the locality of updating. We look into the details in the next section.

5.3 Locality Preservation

Incremental update can be achieved if the locality of an editing function is propagated to the source level. Specifically, we are looking for a subterm in the source that is used for generating the affected subterm in the view, which implies matching of the structures of the source and the view. Figure 5.2 shows the pairing of source/view subterms: the idea is that subterm v_1 of the view depends only on subterm s_1 of the source, $v_2 \lessdot v_1$ only on $s_2 \lessdot s_1$, and so on, and finally $v_n \lessdot ... \lessdot v_2 \lessdot v_1$ only on $s_n \lessdot ... \lessdot s_2 \lessdot s_1$. This kind of locality preservation is apparently determined by the retrieval function, which defines the connection between a view and its source.

5.3.1 Alignment

Definition 5.11 (Alignment) *Given a retrieval function f, we say f aligns at subterm r of s if*

$$\forall t.f\ ((s\ /\ r) \lessdot t) = (f\ s\ /\ f\ r) \lessdot f\ t$$

We call r an alignment position in s with respect to f. □

Very often, when the retrieval function f and source s are known from the context, we simply call r an alignment position, and say f aligns at r. The above definition not only states the matching of source subterms r and view subterm $f\ r$, but also a kind of isolation between them. An alignment position can be seen as a resistive barrier between

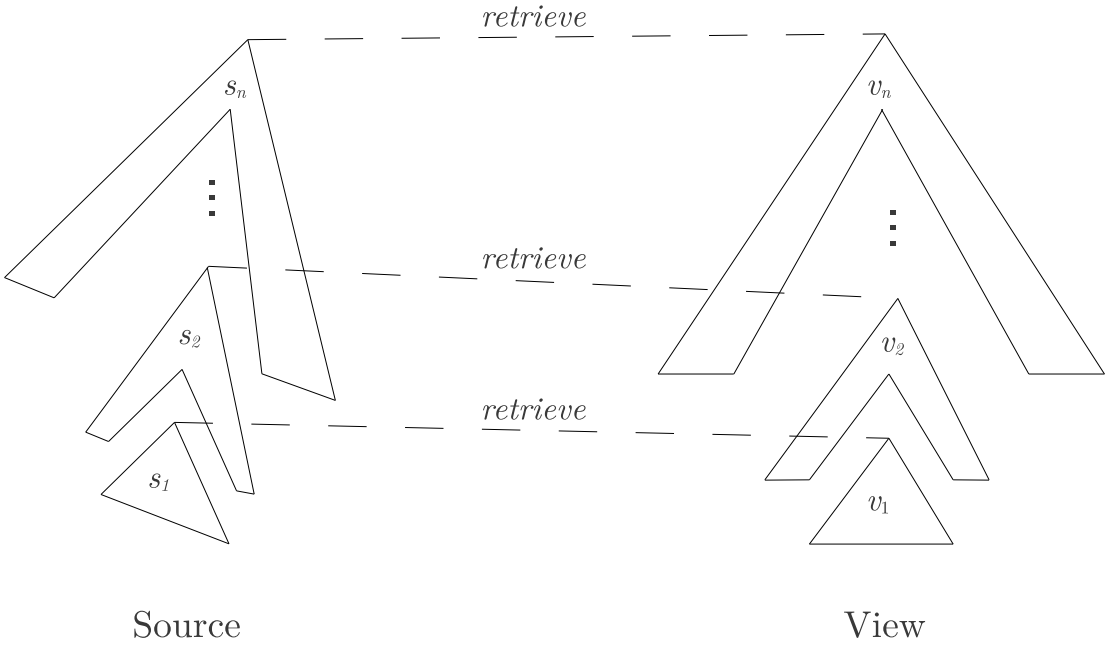


Figure 5.2: Source-View Alignment

the construction of a subterm and its context, where information does not flow through. At an alignment position, $f\ r$ is independent of $s \mathbin{/} r$ and $f\ s \mathbin{/} f\ r$ is independent of t.

The significance of alignment positions is that they capture the mapping between the locality to $f\ r$ in the view and the locality to r in the source. As a result, if $f\ r$ can be locally edited, then r can be locally updated, as shown in the following definition:

$$f^{<}\ ((f\ s \mathbin{/} f\ r) \lessdot v', s) = (s \mathbin{/} r) \lessdot f^{<}_{st}\ (v', r)$$

The above defines a new putback function based on an existing one. Basically, to putback an edited view $(f\ s \mathbin{/} f\ r) \lessdot v'$, we only need to putback v' (the edited $f\ r$), provided f aligns at r. To show that the above transformation is correct, we prove the consistency of $f^{<}$.

$$\begin{array}{ll} & f\ (f^{<}\ ((f\ s \mathbin{/} f\ r) \lessdot v', s)) \\ = & \quad \{\, \text{Definition of } f^{<} \,\} \\ & f\ ((s \mathbin{/} r) \lessdot f^{<}_{st}\ (v', r)) \\ = & \quad \{\, f \text{ aligns at } r \,\} \\ & (f\ s \mathbin{/} f\ r) \lessdot f\ (f^{<}_{st}\ (v', r)) \\ = & \quad \{\, \text{Consistency of } f^{<}_{st} \,\} \\ & (f\ s \mathbin{/} f\ r) \lessdot v' \end{array}$$

Other bidirectional properties hold as well. We postpone proofs of them until Section 5.4, where the complete solution is presented.

Not all view subterms match exactly with a source subterm. Sometimes, we need to resort to a looser fit.

Definition 5.12 *Given a retrieval function f, we say an alignment position r covers v' if $v' \preccurlyeq f\ r$.*

We now show some example transformations that preserve different degrees of alignment. Consider a function that returns the mirror image of a tree.

$$
\begin{aligned}
&\mathit{mirror} :: \mathit{Tree} \to \mathit{Tree}\\
&\mathit{mirror}\ \mathit{Tip} &&= \mathit{Tip}\\
&\mathit{mirror}\ (\mathit{Node}\ a\ l\ r) &&= \mathit{Node}\ a\ (\mathit{mirror}\ r)\ (\mathit{mirror}\ l)
\end{aligned}
$$

Every subtree in the source is an alignment position, because the constructions of the view and of the source coincide. At the other end of the spectrum, consider a function that prunes a tree based on the parity of the sizes of the subtrees.

$$
\begin{aligned}
&\mathit{trim} :: \mathit{Tree} \to \mathit{Tree}\\
&\mathit{trim}\ \mathit{Tip} \quad = \mathit{Tip}\\
&\mathit{trim}\ (\mathit{Node}\ a\ l\ r)\\
&\quad \mid \mathit{even}\ s = \mathit{Node}\ a\ (\mathit{trim}\ l)\ \mathit{Tip}\\
&\quad \mid \mathit{odd}\ s\ \ = \mathit{Node}\ a\ \mathit{Tip}\ (\mathit{trim}\ r)\\
&\qquad \textbf{where}\ s = \mathit{size}\ l + \mathit{size}\ r
\end{aligned}
$$

In this case, no alignment position exists, except the trivial one that takes the complete source, because changes in subtrees propagate up through the structure and cannot be localized.

Another example is a function that extracts the spine of a tree.

$$
\begin{aligned}
&\mathit{spine} :: \mathit{Tree} \to [\mathit{Int}]\\
&\mathit{spine}\ \mathit{Tip} &&= [\,]\\
&\mathit{spine}\ (\mathit{Node}\ a\ l\ r) &&= a : \mathit{spine}\ r
\end{aligned}
$$

In this case, all subterms of the source are alignment positions, though the left-trees, which always correspond to the empty list in the view, are not very interesting.

5.3.2 Exploiting Regularity

It is obvious that change-based putback functions only make sense when there are plenty of alignment positions to choose from, as alignment positions represent matching of source and view construction. For a recursive retrieval function, this suggests a kind of structural recursion pattern. Though not being a sufficient condition, regularity of the recursive pattern is likely to positively impact the availability of alignment positions; and with more alignment positions available, the chance of finding a small but sufficient one increases. Thus, we focus on regular structural recursions – functions that can be implemented as folds. For a regular structural recursion, a source is deconstructed in a uniform way, which leaves the fold body to determine whether the construction of a view matches up.

To explain the intuition behind good alignments, let's revisit the function *spine*. For non-empty input tree, the fold deconstructs it into two source subterms l and r and a single label a, whereas the fold body discards l and adds the label a to *spine* r. The recursive calls, such as *spine* r, always produce a view subcomponent (*spine* r) from a source subterm (r); whether the produced view subcomponent is made into a view subterm by the fold body decides the alignment. For example, subcomponent *spine* r is a subterm of $a : \mathit{spine}\ r$, which makes r an alignment position. In this case, any edit local to *spine* r can be addressed by updating r. In contrast, if we define *spine* as

$$\mathit{spineR}\ (\mathit{Node}\ a\ l\ r) = \mathit{spineR}\ r +\!\!+ [a]$$

then the view construction is the "opposite" of the source construction, with the parent label a at the bottom (end) of the list. This misalignment can be picked out by observing that *spine* r does not form a subterm in the view. Any edit to the view affects a sublist including a, which implies an update to the complete source tree.

In general, for a view subterm to be covered by alignment, it must exclusively originate from a view subcomponent. For example, consider the following variant of *mirror*

$$\begin{aligned}
&\mathit{mirror}'\ (\mathit{Node}\ a\ l\ r)\\
&\quad \mid \mathit{even}\ s = \mathit{Node}\ a\ (\mathit{mirror}'\ r)\ (\mathit{mirror}'\ l)\\
&\quad \mid \mathit{odd}\ s\ \ = \mathit{Node}\ a\ (\mathit{shuffle}\ (\mathit{mirror}'\ r))\ (\mathit{mirror}'\ l)\\
&\qquad \textbf{where}\ s = \mathit{size}\ l
\end{aligned}$$

where *shuffle* shuffles the labels in a tree. The right subtree of the view is the same as $\mathit{mirror}'\ l$ while the left one has neither subcomponents as its exclusive origin. Moving upwards does not help; at any level, the construction of the left subtree requires information

external to the subcomponent $mirror'\ r$. As a result, source subterm l is an alignment position, but not r; and editing to the left part of the view tree cannot be addressed by updating locally in the source.

We formalize the above observation into a condition of fold bodies. We work in the setting of total functions, and call the initial $\mathcal{F}$-algebra of the source datatype in. It is known that there is a deconstructor in° that is in's inverse, which allows us to derive the following evaluation rule from the universal property.

Fact 5.13 (Evaluation Rule) *Given a retrieval of function f with body b and the base functor $\mathcal{F}$ of the source type, we have following evaluation rule of f*

$$f = b \circ \mathcal{F}\ f \circ in^\circ$$

We use a function $arity :: \mathcal{F}\ a \to Int$ to compute the arity of a $\mathcal{F}$-structure by counting the number of recursive components in it. For example, consider a non-empty binary tree t, we have $arity\ (in^\circ\ t) = 2$. We index all the recursive components with unique integers; and selects them through a projection function $select :: \mathcal{F}\ a \to Int \to a$. As a short-hand, we write x_i for $select\ x\ i$. The selection function has a naturality property.

Fact 5.14 (Naturality) *Given a retrieval function f, source s and the base functor $\mathcal{F}$ of the source type,*

$$f\ (in^\circ\ s)_i = (\mathcal{F}\ f\ (in^\circ\ s))_i$$

for any i. □

We write $x[i \mapsto t]$ for substituting x_i by t in x, which binds tighter than function applications. When applied to $\mathcal{F}$-structures, substitution can be related to subtraction of subterms.

Fact 5.15 *Given trees s and t,*

$$in\ (in^\circ\ s)[i \mapsto t] = (s\ /\ (in^\circ\ s)_i) \lessdot t$$

for any i. □

The condition that a retrieval function must satisfy is stated below.

Definition 5.16 (Well-aligning) *We say a retrieval function body b is well-aligning if*

$$\forall u \preccurlyeq b\ x.\exists i.u \sim x_i \wedge \forall w.b\ x[i \mapsto w] = b\ x\ /\ x_i \lessdot w$$

for any x such that arity $x \neq 0$. □

We do not worry about the case when there is no subcomponent in x (i.e.,*arity* $x = 0$), as they are terminals in construction, and will not affect alignment. There are two parts in the condition: the first part ($u \sim x_i$) enforces the matching of the construction of the source with the construction of the view, manifested as an ordering relationship between u and x_i; the second part ($\forall w.b\ x[i \mapsto w] = b\ x\ /\ x_i \lessdot w$)) restricts b to be non-strict in any subcomponents that match view subterms, so that they can be changed without affecting the execution of b. It is important that the non-strictness requirement only applies to selected subcomponents; some, that are sufficient to cover all the subterms of $b\ x$, are taken as opaque blocks while leaving the rest to be broken up for gluing the blocks.

Let's look at a few more examples.

$$\begin{aligned}
&g1\ (x, xs, ys) = [x] +\!\!+ xs +\!\!+ ys\\
&g2\ (x, xs, ys) = xs +\!\!+ [x] +\!\!+ ys\\
&g3\ (y, xs, ys) = xs +\!\!+ ys +\!\!+ [y]
\end{aligned}$$

These three functions correspond to individual cases of the fold bodies for traversing binary trees in pre-, in-, and post-order respectively. There are two inputs to the functions that are view subcomponents, namely *xs* and *ys*. Functions *g1* and *g2* are well-aligning, *ys* is ordered with all the view subterms, whereas *g3* is not.

Having the view subcomponents and view subterms ordered enforces that the constructions of source and view are in the same "direction", but not the independence of the subcomponents. Consider the following function that keeps one list of two, based on the size of one of them.

$$\begin{aligned}
&g4\ (x, xs, ys)\\
&\quad \mid even\ (length\ xs) = [x] +\!\!+ ys\\
&\quad \mid otherwise\ \ \ \ \ \ \ \ = [x] +\!\!+ xs
\end{aligned}$$

In this case, though all the view subterms are ordered with some view subcomponent, it is wrong to conclude that edits to subterms *xs* and *ys* can be dealt with locally. If the size of *xs* changes, function *g4* may behave differently.

On the other side of the information flow, we do not allow the computation of a subcomponent to be influenced by its context either. Consider the following example.

$$g5\ (x, xs, ys) = xs +\!\!+ [x] +\!\!+ reverse\ ys$$

The subcomponent *ys* is changed by the function; and its manifestation in the view depends on its context, which decides how many times *reverse* is applied to it. In this case, though edits remain local to the subcomponent, it is difficult to reconstruct its source, since we don't know exactly what transformation the subcomponent has gone through. For these reasons, both *g4* and *g5* won't form well-aligning fold bodies for binary trees; and the well-aligning condition is designed to rule out both the above scenarios.

Generalizing the definition to the semantics of transformations, we say that a retrieval function is well-aligning if all cases of its body are well-aligning: *preorder*, *inorder*, *unzip*, *mirror*, *spine*, *filter*, *map* etc. are examples of well-aligning retrieval functions, while *postorder*[1] and *trim* are not.

The well-aligning property guarantees the availibility of alignment positions; and we can state a declarative result about how they may be found.

Theorem 5.17 *Given a well-aligning retrieval function f such that $f\ s = v$, we have that f aligns at subterm r of s if there exists a non-trivial subterm u of v such that $u \preccurlyeq f\ r$.* □

The well-aligning condition tells us clearly that some selected subcomponents become subterms in the view; and the source subterms producing the selected subcomponents are alignment positions. The key to proving Theorem 5.17 is to establish the fact that subcomponent $f\ r$ is among those selected due to the premise $u \preccurlyeq f\ r$; this can be achieved by connecting the unique labels in u with those in $f\ r$. As preparation for formally proving Theorem 5.17, we state some properties regarding labels of trees under transformation.

We use *Labels* as the type of label sets. A function

$$lab_{\mathcal{F}} :: \mathcal{F}\ Labels \to Labels$$

unions all label sets in a $\mathcal{F}$-structure. Note that $lab_{\mathcal{F}}$ forms an algebra, so that the label extraction function *label* (also written as $\langle\cdot\rangle$) can be defined in term of it: $label_{\mu\mathcal{F}} =$

[1] We will discuss how this function can be made well-aligning in Section 5.5.

fold $lab_{\mathcal{F}}$. We often omit the type subscript when it is clear from the context, and only use *lab* and *label*. The function *lab* has the following properties (where as a shorthand, we write '$\langle x_{\neq i} \rangle$' for '*lab* ($\mathcal{F}$ *labels* x)$[i \mapsto \emptyset]$').

Fact 5.18 (Union) *Given* $x :: \mathcal{F}\ \mu\mathcal{G}$*, for any* i*,*

$$\langle x_{\neq i} \rangle \cup \langle x_i \rangle = \langle x \rangle$$

□

Fact 5.19 (Disjoint) *Given* $s :: \mu\mathcal{F}$*, for any* i*,*

$$\langle (in^\circ\ s)_{\neq i} \rangle \# \langle (in^\circ\ s)_i \rangle$$

where $x \# y$ *denotes that* $x \cap y = \emptyset$*.*

□

(Note that $(x \subseteq y) \wedge (y \# z) \Rightarrow (x \# z)$ by monotonicity of intersection, so we allow ourselves to write derivations of the form "$w \subseteq x \# y \supseteq z$", with a chain of inclusions, a disjointness, and a chain of containments, and conclude that $w \# z$.)

As mentioned at the beginning of Section 5.2, one important assumption about retrieval functions is that they do not invent labels.

Requirement 5.20 (Non-invention of Labels) *Given a retrieval function* f *with body* b *and the base functor* $\mathcal{F}$ *of the source type, we have*

$$\forall x. \langle b\ x \rangle \subseteq \langle x \rangle$$

and

$$\forall s. \langle f\ s \rangle \subseteq \langle s \rangle$$

□

An important consequence of the fact that retrieval functions do not invent labels is that labels cannot reappear after they have been dropped during the construction of a view. Conversely, if a label set $\langle v \rangle$ has been generated after processing subterm r of t by the retrieval function f (that is, $r \preccurlyeq t$ and $\langle v \rangle \subseteq \langle f\ r \rangle$), and $\langle v \rangle$ is still present after processing t itself (that is, $\langle v \rangle \subseteq \langle f\ t \rangle$), then $\langle v \rangle$ is present at every intermediate stage too ($\langle v \rangle \subseteq \langle f\ s \rangle$ for every s such that $r \preccurlyeq s \preccurlyeq t$. This is a kind of "convexity" property of label sets.

More importantly in what follows, a similar result holds for subterms, rather than their projections to label sets; but for this, we need the additional assumption that the retrieval function f is well-aligning. The primary result (Corollary 5.22) is a convexity property for terms: given sources r, t with $r \preccurlyeq t$ such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ s$ for any s such that $r \preccurlyeq s \preccurlyeq t$. (In fact, $v \preccurlyeq f\ t$ is not strictly required; $\langle v \rangle \subseteq \langle f\ t \rangle$ suffices.) The essential step (Lemma 5.21) is the one from the outermost term t to one of its immediate children: if view subterm v shows up after processing a subterm r within the ith child $(in^\circ\ t)_i$ of t, and v is still present after processing t, then v must have come from the ith child: $v \preccurlyeq f\ (in^\circ\ t)_i$. Note that, for both these results, we make use of our implicit assumption that v is non-trivial.

Lemma 5.21 (Maintaining terms) *Suppose a well-aligning retrieval function $f = fold\ b$. For source terms r, t with $r \preccurlyeq (in^\circ\ t)_i$, if $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ (in^\circ\ t)_i$.*

Proof:
Let $x = \mathcal{F}\ f\ (in^\circ\ t)$, so that $f\ t = b\ x$ and $f\ (in^\circ\ t)_i = x_i$. Since b is well-aligning, and v is a non-trivial subterm of $b\ x$, there exists a j such that $v \sim x_j = f\ (in^\circ\ t)_j$. In fact, this j must be i:

$$
\begin{array}{ll}
 & \langle v \rangle \\
\subseteq & \quad \{\, \text{labels of subterms (Fact 5.4); } v \preccurlyeq f\ r \text{, by assumption} \,\} \\
 & \langle f\ r \rangle \\
\subseteq & \quad \{\, f \text{ does not invent labels} \,\} \\
 & \langle r \rangle \\
\subseteq & \quad \{\, r \preccurlyeq (in^\circ\ t)_i \text{; Fact 5.4 again} \,\} \\
 & \langle (in^\circ\ t)_i \rangle \\
\# & \quad \{\, \text{disjointness of labels} \,\} \\
 & \langle (in^\circ\ t)_{\neq i} \rangle \\
\supseteq & \quad \{\, f \text{ does not invent labels; monotonicity of intersection} \,\} \\
 & \langle (\mathcal{F}\ f\ (in^\circ\ t))_{\neq i} \rangle \\
= & \quad \{\, \text{definition of } x \,\} \\
 & \langle x_{\neq i} \rangle
\end{array}
$$

and so $\langle v \rangle \# \langle x_{\neq i} \rangle$, and hence $\langle v \rangle \subseteq \langle x_i \rangle$ by disjointness of labels. Finally, $v \sim x_i$ and $\langle v \rangle \subseteq \langle x_i \rangle$ imply $v \preccurlyeq x_i$, by Fact 5.4. □

Corollary 5.22 (Term convexity) *Suppose a well-aligning retrieval function $f = fold\ b$. For source terms r, t with $r \preccurlyeq t$, if $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ s$ for every s such that $r \preccurlyeq s \preccurlyeq t$.*

Proof:
Proof by induction over the length of *location* $t\ r$. The base case is when the path is empty, so $r = t$; then the lemma is trivially true. For the inductive case, assume the statement is valid for paths of length n. Suppose that $r \preccurlyeq t$, and that r is at depth $n+1$ in t (so that $r \preccurlyeq (in^\circ\ t)_i$ for some unique index i, and *location* $(in^\circ\ t)_i\ r$ has length n), and that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$. By Lemma 5.21, we get $v \preccurlyeq f\ (in^\circ\ t)_i$; then by induction we get $v \preccurlyeq f\ s$ for every s with $r \preccurlyeq s \preccurlyeq (in^\circ\ t)_i$ too; and the final case $s = t$ trivially holds. □

Theorem 5.17 follows directly from the following result: given a well-aligning *get* function f, and sources r,s such that $r \preccurlyeq s$, if there exists any view subterm v such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ s$, then f aligns at subterm r of s. (Again, we assume that v is non-trivial.)

Lemma 5.23 (Get alignment) *Given a well-aligning retrieval function f, sources r, t with $r \preccurlyeq t$, and view v such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then f aligns at subterm r of t.*

Proof:
Proof by induction over the length of *location* $t\ r$. The base case is when the path is empty; then $t = r$ and the theorem is trivially true (since f necessarily aligns at the root t of source t). For the inductive case, we assume that the statement is valid for paths of length n; we are given terms r, t with $r \preccurlyeq t$ and r at depth $n+1$ in t, and a term v with $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, and we have to show that f aligns at subterm r of t.

Suppose that r is within the i'th child of t, that is, $r \preccurlyeq s$ where $s = (in^\circ\ t)_i$. Then by Corollary 5.22, we have $v \preccurlyeq f\ s$, and by induction, f aligns at subterm r of s. Let $x = \mathcal{F}\ f\ (in^\circ\ t)$, so that $f\ t = b\ x$ and $f\ (in^\circ\ t)_i = x_i$. Because b is well-aligning and $v \preccurlyeq b\ x$, there exists a j such that $v \sim x_j$ and $b\ x[j \mapsto w] = b\ x \,/\, x_j \lessdot w$ for any w. In fact, that j must be i, by the same argument as in the proof of Lemma 5.21. In particular, $f\ s = x_i \preccurlyeq b\ x$, a fact that we shall use below. Finally, we show that f aligns at subterm r of t. For an arbitrary source term p, we have:

$$
\begin{aligned}
& f\ (t \,/\, r \lessdot p) \\
= \quad & \{\text{since } r \preccurlyeq s \preccurlyeq t\} \\
& f\ (t \,/\, s \lessdot (s \,/\, r \lessdot p))
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{\,\text{Fact 5.15—}s = (in^\circ\ t)_i\,\} \\
& f\ (in\ (in^\circ\ t)[i \mapsto (s\ /\ r \lessdot p)]) \\
= & \{\,\text{evaluation rule for } f = \mathit{fold}\ b\,\} \\
& b\ (\mathcal{F}\ f\ (in^\circ\ t)[i \mapsto (s\ /\ r \lessdot p)]) \\
= & \{\,\text{naturality of } \mathit{select}\,\} \\
& b\ (\mathcal{F}\ f\ (in^\circ\ t))[i \mapsto (f\ (s\ /\ r \lessdot p))] \\
= & \{\,b \text{ is well-aligning; discussion above}\,\} \\
& b\ (\mathcal{F}\ f\ (in^\circ\ t))\ /\ f\ (in^\circ\ t)_i \lessdot f\ (s\ /\ r \lessdot p) \\
= & \{\,\text{evaluation rule for } f \text{ again; } s = (in^\circ\ t)_i\,\} \\
& f\ t\ /\ f\ s \lessdot f\ (s\ /\ r \lessdot p) \\
= & \{\,\text{induction}\,\} \\
& f\ t\ /\ f\ s \lessdot (f\ s\ /\ f\ r \lessdot f\ p) \\
= & \{\,\text{nesting}\,\} \\
& f\ t\ /\ f\ r \lessdot f\ p
\end{array}
$$

□

So far, we have established well-aligning as a sufficient condition for the availability of alignment (Definition 5.16), and a declarative result about how alignment positions can be found (Theorem 5.17). Next, we move on to devise a constructive method of finding alignment positions, and deriving a change-based putback function based on this method.

5.4 Change-Based Putback Functions

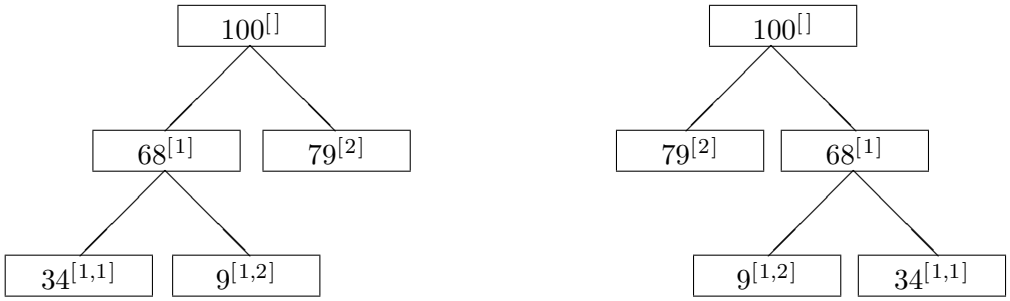
Our derivation of putback functions is divided into three steps: (i) finding an alignment position covering the edited view subterm; (ii) using a state-based putback function to map the edited view subterm to a source subterm; (iii) merging the original source context with the updated source subterm.

Step (i) is the key, while the other two follow straightforwardly. Taking the previous result, we know a source subterm is an alignment position if there is a corresponding subterm in the view. A standard way of establishing the source/view correspondence semantically is to trace the uniquely identified labels.

5.4.1 Indexing and Reflecting

We index source labels using paths from the root to the constructors that the labels are attached to. As a result, a label is at the root position of the subterm identified by

the path. The indices are not separable from the labels and are moved together by a retrieval function. As a result, a label in the view originates from a label in the source with the same index. (It is worth noting that the indices only represent paths in the source, not those in the view.) The indices of the labels serve as unique identifiers, so $\langle x \rangle$ denotes all the indices in x. Given an edit-affected view subterm v, a sensible alignment position should include all indices in $\langle v \rangle$; the path leading to such a source subterm is the maximum common prefix of all the paths, *mcp* $\langle v \rangle$. Consider a simple example with *mirror* as the retrieval function:



Source Tree View Tree

Note that the indices in the source – the superscript lists of numbers in the diagram – are copied over to the view. Suppose we insert a node at the index position $[1]$ in the view, affecting the element 68 at position $[1]$ and those below it $[[1,1],[1,2]]$. (Note that we don't require a concrete index for the newly inserted node since it won't contribute to the identification of the affected source.) The maximum common prefix of $[[1],[1,1],[1,2]]$ is $[1]$. Now taking the path $[1]$ back to the source, we conclude that it is the subtree with root 68 that needs to be changed.

The key to demonstrating the correctness of the above process is to show that the subset relation between index sets corresponds to the subterm relation between trees.

Lemma 5.24 *Given a well-aligning retrieval function $f = fold\ b$, and source terms s, t with $s \preccurlyeq t$, and view term v, if $v \preccurlyeq f\ t$ and $\langle v \rangle \subseteq \langle s \rangle$ then $v \preccurlyeq f\ s$.*

Proof:
Proof by induction over the length of *location* $t\ s$. The base case is when the path is empty; then $s = t$, and the result trivially holds. For the inductive case, assume that the result holds for paths of length n, and that s is at depth $n + 1$ in t. Let i be such that

$s \preccurlyeq (in^\circ\ t)_i$, so that *location* $(in^\circ\ t)_i\ s$ has length n. We will show that $v \preccurlyeq f\ (in^\circ\ t)_i$; then we can conclude $v \preccurlyeq f\ s$ by appeal to the inductive hypothesis.

Let $x = \mathcal{F}\ f\ (in^\circ\ t)$, so $f\ t = b\ x$ and $f\ (in^\circ\ t)_i = x_i$. Since v is a non-trivial subterm of $b\ x$, and b is well-aligning, there exists a j such that $v \sim x_j$. By the usual argument, that j must be i:

$$
\begin{array}{ll}
 & \langle v \rangle \\
\subseteq & \quad \{\, \text{assumption} \,\} \\
 & \langle s \rangle \\
\subseteq & \quad \{\, \text{hypothesis, and Fact 5.4} \,\} \\
 & \langle (in^\circ\ t)_i \rangle \\
\# & \quad \{\, \text{disjointness of labels} \,\} \\
 & \langle (in^\circ\ t)_{\neq i} \rangle \\
\supseteq & \quad \{\, f \text{ does not invent labels} \,\} \\
 & \langle (\mathcal{F}\ f\ (in^\circ\ t))_{\neq i} \rangle \\
= & \quad \{\, \text{definition} \,\} \\
 & \langle x_{\neq i} \rangle
\end{array}
$$

So $v \sim x_i$. Moreover, $\langle v \rangle \subseteq \langle x_i \rangle$, by disjointness of labels, since by assumption we have $v \preccurlyeq f\ t$ and hence $\langle v \rangle \subseteq \langle f\ t \rangle = \langle b\ x \rangle \subseteq \langle x \rangle$, and we have just shown that $\langle v \rangle \# \langle x_{\neq i} \rangle$. Therefore, by Fact 5.4 we conclude $v \preccurlyeq x_i = f\ (in^\circ\ t)_i$. □

As a remark, so far we have been oblivious to the fact that the source and view labels are now indexed, and have assumed that the retrieval and putback functions work uniformly on them. This is certainly true if the retrieval function is a natural transformation, like most of our examples, which does not scrutiny the labels. For non-natural-transformations, some adjustments are needed to migrate the state-based bidirectional system to handle indexed labels. However, we expect such adjustments are minimum.

5.4.2 The Change-Based Putback Function

We are now ready to present the change-based putback function. For any $f^{\leq}_{st}$, a generic $f^{\leq}_{ch}$ function can be defined as follows.

$$
\begin{array}{l}
f^{\leq}_{ch} :: \mathit{Edit}\ v \to s \to s \\
f^{\leq}_{ch}\ \mathit{edt}\ s = (s \,/\, r) \lessdot (f^{\leq}_{st} \circ ((\mathit{edit}\ \mathit{edt} \circ f) \vartriangle \mathit{id}))\ r \\
\quad \textbf{where}\ i = \mathit{mcp}\ \langle \mathit{affect}\ \mathit{edt}\ (f\ s) \rangle \quad \text{-- consolidating indices} \\
\qquad\quad r = \mathit{zoom}\ i\ s \qquad\qquad\qquad\quad \text{-- exposing alignment position}
\end{array}
$$

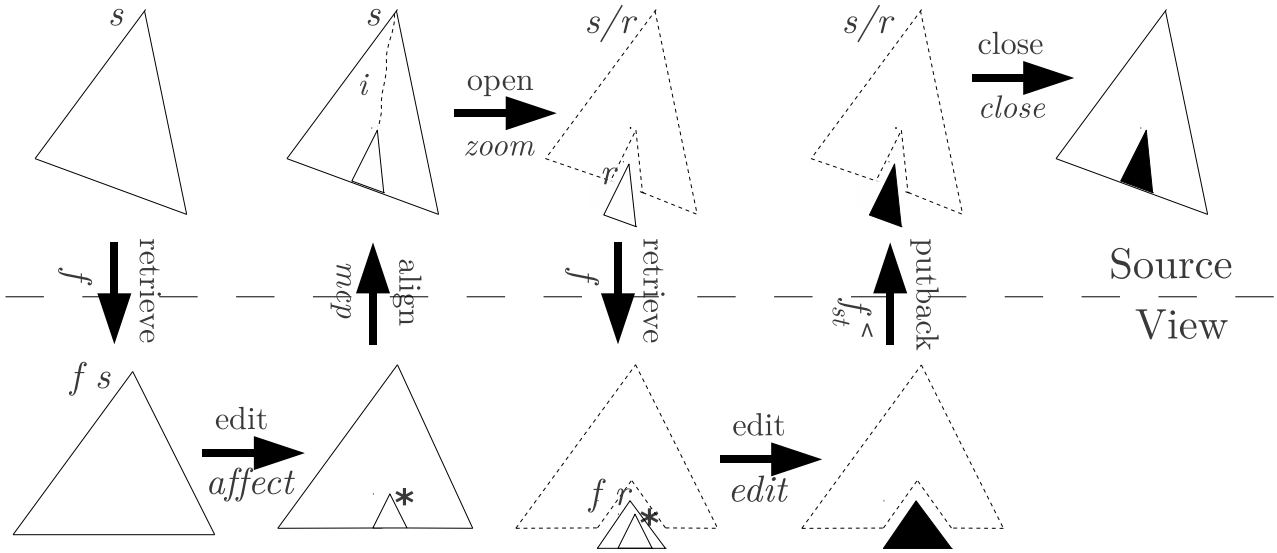


Figure 5.3: Operation-based Putback

Function $f_{ch}^{<}$ maps an operation on views into an operation on sources. There are several steps in this process, which are extracted into **where** clauses. A source is firstly retrieved into a view (via f), with the affected view subterm of the edit extracted (via *affect edt*). After that, the indices in the affected view subterm are collected and are used to identify an alignment position (r), covering the affected view subterm. A view of the alignment position is then constructed by applying f, and is edited before the state-based putback function $f_{st}^{<}$ maps it into an updated source subterm. The standard split function $\triangle$ has type $(a \to b) \to (a \to c) \to (a \to (b, c))$. Finally, we combine the newly created source subterm with the original context that remains unchanged. (Note that we omit the straightforward re-indexing of the newly generated source subterm with i as the root index.) This computation flow is illustrated in Figure 5.3, which shows how an updated source is achieved through an indirect route. There are several passes across the source/view boundary; but importantly other than the initial retrieval, all of them concern only the edit-affected subterms. Assuming editing and retrieval have no worse run-time performance than putting back, the complexity of $f_{ch}^{<}$ is $O\ (m\ \times\ log\ n + c\ m)$ where n and m are the size of the source (s) and the change (r) respectively; and c is the complexity function for $f_{ch}^{<}$. Note that *affect edt* $(f\ s)$ should have been executed prior to the putback execution, and is not included in the performance analysis. We also don't consider the cost of computing $s\ /\ r$ because in a real implementation, the context $s\ /\ r$ can be computed together with the focus r when the zooming is performed. The

$m \times \mathit{log}\ n$ part of the above complexity function comes from the computation of *mpc*, where m indices of size $\mathit{log}\ n$ need to be processed.

Function $f^{<}_{ch}$ is expected to preserve the bidirectional properties of $f^{<}_{st}$. To establish this, we need to know that it does operate at alignment positions.

Theorem 5.25 *Given a well-aligning get function f such that $f\ s = v$, then for all source subterms r of s and view subterms u of v, zoom $(mcp\ \langle u \rangle)\ s$ is the smallest alignment position covering u.*

Proof:
By definition, we have $\langle u \rangle \subseteq \langle \mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s \rangle$.

$$
\begin{array}{ll}
 & \langle u \rangle \subseteq \langle \mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s \rangle \wedge u \preccurlyeq f\ s \wedge \\
 & (\mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s) \preccurlyeq s \\
\Rightarrow & \quad \{\, \text{Lemma 5.24} \,\} \\
 & u \preccurlyeq f\ (\mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s) \wedge u \preccurlyeq f\ s \wedge \\
 & \langle \mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s \rangle \preccurlyeq s \\
\Rightarrow & \quad \{\, \text{Theorem 5.17} \,\} \\
 & f \text{ aligns at subterm } \mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s \text{ of } s
\end{array}
$$

Also from the definition of *mcp*, there exists no r such that $r \prec \mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s$ and $\langle u \rangle \subseteq \langle r \rangle$. Thus, $\mathit{zoom}\ (\mathit{mcp}\ \langle u \rangle)\ s$ is the smallest alignment position covering u.

□

We can state the bidirectional properties of $f^{<}_{ch}$.

Theorem 5.26 (Consistency) $f\ (f^{<}_{ch}\ \mathit{edt}\ s) = \mathit{edit}\ \mathit{edt}\ (f\ s)$

Proof:

$$
\begin{array}{ll}
 & f\ (f^{<}_{ch}\ \mathit{edt}\ s) \\
= & \quad \{\, \text{Definition of } f^{<}_{ch} \,\} \\
 & f\ (s\ /\ r \lessdot (f^{<}_{st} \circ ((\mathit{edit}\ \mathit{edt} \circ f)\ \vartriangle\ \mathit{id}))\ r)) \\
= & \quad \{\, r \text{ is an alignment position} \,\} \\
 & f\ s\ /\ f\ r \lessdot (f \circ f^{<}_{st} \circ ((\mathit{edit}\ \mathit{edt} \circ f)\ \vartriangle\ \mathit{id}))\ r) \\
= & \quad \{\, \text{Consistency of } f^{<}_{st} \,\} \\
 & f\ s\ /\ f\ r \lessdot (\mathit{edit}\ \mathit{edt} \circ f)\ r \\
= & \quad \{\, r \text{ covers } \mathit{affect}\ \mathit{edt}\ (f\ s), \text{ and locality of } \mathit{edit} \,\} \\
 & \mathit{edit}\ \mathit{edt}\ (f\ s\ /\ f\ r \lessdot f\ r)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\, r \text{ is an alignment position} \,\} \\
& \mathit{edit}\ \mathit{edt}\ (f\ (s \mathbin{/} r \lessdot r))) \\
= & \{\, \text{Cancellation} \,\} \\
& \mathit{edit}\ \mathit{edt}\ (f\ s)
\end{array}
$$

□

For acceptability, we need an "identity" edit that does not change the view.

Theorem 5.27 (Acceptability) $f^{<}_{ch}\ (E\ \{\, \mathit{edit} = \mathit{id} \,\}) = \mathit{id}$.

Proof:

$$
\begin{array}{ll}
& f^{<}_{ch}\ (E\ \{\, \mathit{edit} = \mathit{id} \,\})\ s \\
= & \{\, \text{Definition of } f^{<}_{ch} \,\} \\
& s \mathbin{/} r \lessdot (f^{<}_{st} \circ ((\mathit{id} \circ f)\ \vartriangle\ \mathit{id}))\ r \\
= & \{\, \text{Acceptability of } f^{<}_{st} \,\} \\
& s \mathbin{/} r \lessdot r \\
= & \{\, \text{Cancellation} \,\} \\
& s
\end{array}
$$

□

Undoability involves inverting an edit as a function.

Theorem 5.28 (Undoability) $f^{<}_{ch}\ (\mathit{edt}\ \{\, \mathit{edit} = (\mathit{edit}\ \mathit{edt})^{-1} \,\}) \circ f^{<}_{ch}\ \mathit{edt} = \mathit{id}$.

Proof:

$$
\begin{array}{ll}
& (f^{<}_{ch}\ (\mathit{edt}\ \{\, \mathit{edit} = (\mathit{edit}\ \mathit{edt})^{-1} \,\}) \circ f^{<}_{ch}\ \mathit{edt})\ s \\
= & \{\, \text{Definition of } f^{<}_{ch}, \text{ and constant } \mathit{affect} \text{ in } \mathit{edt} \,\} \\
& s \mathbin{/} r \lessdot (f^{<}_{st} \circ ((\mathit{edit}\ \mathit{edt})^{-1} \circ f\ \vartriangle\ \mathit{id}))\ ((f^{<}_{st} \circ (\mathit{edit}\ \mathit{edt} \circ f\ \vartriangle\ \mathit{id}))\ r) \\
= & \{\, \text{Definition of } \vartriangle \,\} \\
& s \mathbin{/} r \lessdot f^{<}_{st}\ (((\mathit{edit}\ \mathit{edt})^{-1} \circ f \circ f^{<}_{st})\ ((\mathit{edit}\ \mathit{edt} \circ f)\ r, r), \\
& \qquad\qquad f^{<}_{st}\ ((\mathit{edit}\ \mathit{edt} \circ f)\ r, r)) \\
= & \{\, \text{Consistency of } f^{<}_{st} \,\} \\
& s \mathbin{/} r \lessdot f^{<}_{st}\ (((\mathit{edit}\ \mathit{edt})^{-1} \circ \mathit{edit}\ \mathit{edt} \circ f)\ r), f^{<}_{st}\ ((\mathit{edit}\ \mathit{edt} \circ f)\ r, r)) \\
= & \{\, (\mathit{edit}\ \mathit{edt})^{-1} \circ \mathit{edit}\ \mathit{edt} = \mathit{id} \,\} \\
& s \mathbin{/} r \lessdot f^{<}_{st}\ (f\ r, f^{<}_{st}\ ((\mathit{edit}\ \mathit{edt} \circ f)\ r, r)) \\
= & \{\, \text{Undoability of } f^{<}_{st} \,\} \\
& s \mathbin{/} r \lessdot r
\end{array}
$$

$$= \quad \{ \text{Cancellation} \}$$
$$s$$

□

5.5 More Refined Locality

As we have seen, the performance of our proposal depends on the height of the source tree and the size of the affected region (i.e., the level of locality of the editing system). The former is clearly beyond the control of any bidirectional system; while the latter is again largely decided by the structure of the view. For fairly balanced trees, the majority of nodes are deep in the structure, so it is reasonable to suppose that the majority of edits will be too; given structure alignment, this implies a good degree of locality. A problem arises when the view tree is skewed, say being a list; then the likelihood that a node appears at any depth is the same. If a node high in the structure is affected by an edit, say deleted, the affected subtree could be rather large.

This problem has already manifested itself in our binary-tree traversal example (see Section 5.1.1, where marking the whole sublist $[40, 58, 9]$ as affected is excessive). A better alternative is to recognize the sublist $[58, 9]$ as unaffected context too.

Another example is post-order tree traversal where any non-empty sublist of the view contains the head of the source, which results in very poor locality preservation. As a matter of fact, post-order traversal is excluded through the well-aligning condition.

Yet, being a special kind of tree, lists enjoy a number of unique properties. We notice that unlike general trees, where a separate datatype is needed for contexts, we can simply use lists as both contexts and focuses, and use the append function $+\!\!+$ as the close function. Given the symmetry of $+\!\!+$, either the context or the focus can be edited, and all the definitions and results dualize. For example, consider the *reverse* function. Editing the front of the list view can be localized to a prefix of the view and mapped back to a suffix of the source.

As a result, it makes sense to try to capture a lower (right) bound of an edit-affected sublist, in addition to the upper (left) bound. Instead of splitting a list view into a prefix (context) and a suffix (focus), we can now see it as $l1 +\!\!+ l2 +\!\!+ l3$. To reflect this specialization, we overload the infix operator $\lessdot$ and define its list version as $(l1\,, l3) \lessdot l2 = l1 +\!\!+ l2 +\!\!+ l3$. All the other definitions developed for general trees remain valid.

Now instead of always picking out a complete suffix or prefix, we can mark a middle segment as affected by editing, and the same $f_{ch}^{<}$ directly applies. For example, deleting 40 from $[7, 36, 100, 40, 58, 9]$ only affects the middle segment $[40]$ while leaving both contexts $[7, 36, 100]$ and $[58, 9]$ unaffected. Correspondingly, the alignment positions now match subterms in the source to segments (rather than tails) in the view.

5.6 Discussion

5.6.1 Context-Sensitive Editing

The editing system we have looked at so far is context-independent, which is particularly convenient for local editing since the same edit can be applied both to a structure and its subterms. For tree-structured views, it is sometimes useful to provide a full (or partial) path to the intended editing location to narrow down the search. In this case, the editing becomes context sensitive because the starting point of the path matters. Consider a path-based editing system.

$$\textbf{type } \mathit{EditP}\ t = \{\, \mathit{edit} :: \mathit{Path} \to t \to t, \mathit{affect} :: \mathit{Path} \to t \to \mathit{Path} \,\}$$

An edit operation now finds its target in a structure following a path, and produces the edited structure together with the path leading to the affected subterm. Note that these paths in the view should not be confused with indices of labels that represent paths in the source.

The definition of $f_{ch}^{<}$ can be adapted for the new editing system. We separate all interesting steps into **where** clauses to facilitate explanation.

$$\begin{aligned}
&f_{ch}^{<} :: \mathit{Edit}\ v \to \mathit{Path} \to s \to s \\
&f_{ch}^{<}\ \mathit{edt}\ p\ s = (s\ /\ r) \lessdot f_{st}^{<}\ (\mathit{edit}\ \mathit{edt}\ \mathit{p3}\ u', r) \\
&\quad \textbf{where } v = f\ s \\
&\qquad\quad \mathit{p1} = \mathit{affect}\ \mathit{edt}\ p\ v \\
&\qquad\quad u = \mathit{zoom}\ \mathit{p1}\ v \qquad\qquad \text{-- finding affected subterm} \\
&\qquad\quad i = \mathit{mcp}\ \langle u \rangle \\
&\qquad\quad r = \mathit{zoom}\ i\ s \\
&\qquad\quad u' = f\ r \\
&\qquad\quad \mathit{p2} = \mathit{travelUntil}\ (\mathit{p1}, v)\ u' \qquad \text{-- finding path to } u' \\
&\qquad\quad \mathit{Just}\ \mathit{p3} = \mathit{stripPrefix}\ \mathit{p2}\ \mathit{p1} \qquad \text{-- finding relative editing path}
\end{aligned}$$

Note that we keep the path information of the edit explicit so that it can be modified along with the shifting of focus. Compared with the context-independent version, there are a few additional steps. As the editing function returns a path locating the affected subterm, we need to open the view to get to it. A bigger challenge posed by this context-sensitivity is to find a relative editing path when the starting point is moved to the root of subterm u'. We denote the path from the root of a structure x to its subterm y as $x \longrightarrow y$. Since we know the subterm relations among the affected subterm, retrieval result of the alignment position and the view $u \preccurlyeq u' \preccurlyeq v$, the path $u' \longrightarrow u$ (*p3* above), is the difference between $v \longrightarrow u$ (*p1* above) and $v \longrightarrow u'$ (*p2* above). We already know *p1*; traversing *p1* until the root of u' gives us *p2*, before we can perform path arithmetic to recover the correspondence between the path and structure inputs of *edit edt*. Function *travelUntil* follows a path down a tree until it reaches a given subtree; the part of the path travelled is returned as the output. Function *stripPrefix* is a standard Haskell function of type $Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Maybe\ [a]$ that strips the first input from the second one; since we know that *p2* is a prefix of *p1*, the execution of *stripPrefix p2 p1* is always going to succeed. A concern here is that the additional computation does place a performance overhead: the path travelling takes time linear in the height of the view tree. It is obvious that the multiple traversals in the above code can be combined. Nevertheless, we present them in separate steps for clarity.

5.6.2 Connection with Other Bidirectional Approaches

The framework proposed in this chapter is very general. Most existing systems could benefit from it, as long as the retrieval function satisfies the well-aligning construction condition (Definition 5.16). Note that the condition is semantic; there is no restriction on the language that is used for implementation.

The generality we have achieved reflects the fundamentality of bidirectional programming: it is not so much a business of cleverly inventing the new, but one of retaining the old. This principle is made explicit by the constant complement approach, which in essence is a strategy of recovering the original. Prior to this work, a complement is decided by the source and the retrieval function; and the effect of an edit does not come into the picture. We make complements in some sense "edit-sensitive", where the context, being constant, is derived from individual edits. Compared to the original constant complement approach [MHN+07, Chapter 4], the system in this chapter does not make the technique more applicable, since the constant context that is preserved can always

be regenerated by the putback function constructed using the technique in Chapter 4. However, it is clear that this regeneration is a wasted effort, given that the context is a chunk of the source that is not changed. Moreover, by having a separately marked constant context, we can derive a good estimation of the scale of the change as a result of an update, something that has never been easy in other approaches.

Incremental updates have been studied in the context of model transformation, for improving speed [GW09], or for more refined semantics [DXC10]. Similar to our design, they also require additional specification of the effect of an edit. In contrast to tree like datatypes, models are loosely-connected untyped graphs, which are more easily divided into independent fragments to be updated separately; whereas our well-aligning property of typed and overlapping subtrees is much harder to establish. In [HHI+10], where graph transformations are defined as structural recursion, a simplified assumption that different parts of the graph are always independent is used. As a result, the structural recursion is effectively reduced to a map operation. However, this assumption is not true in general; and as a result, the acceptability property does not hold in their system.

Despite in a unidirectional setting, the concept of *adaptive programming* [ABH06] is closely related to incremental updates. The basic idea of adaptive programming is to build up a complete input-output dependency graph for a given input, from some syntactic annotations to the program. Based on the dependency, a corresponding output change can be derived from an input change, which hopefully has a much better run-time performance compared to re-executing the program with the new input. However, it is not obvious how the technique can be applied to a bidirectional setting, where we need to derive an input change from an output change. Nevertheless, it will be an interesting future direction to explore.

Chapter 6

Conclusion

6.1 Summary

Looking back to the challenge laid down at the beginning of this thesis, we have shown with concrete examples the wider applicability of bidirectional programming, and demonstrated the important role played by the edit information in the designing of bidirectional languages. The three cases we have investigated in this thesis are entirely novel: to the best of our knowledge, no existing work has utilized bidirectional techniques in similar ways. Yet, the results are promising; the proposed bidirectional systems naturally fit as solutions to the problems.

In Chapter 3, we tackled the long standing problem of marrying pattern matching with abstraction. Prior to our work, the correctness of such systems is only attempted through explicit proofs of program properties, which is laborious and potentially unreliable. Our work of reasoning about abstract datatypes though their specifications reveals that right-invertibility of the conversion functions is the key to soundness and efficiency; and the use of the right-invertible language RINV discharges any related proof obligation. This result goes beyond being an effective solution to reasoning with abstraction, by offering fresh insights to the development of bidirectional programming. Notably, the "editing" mechanism of views in this application is different from the others': specifications (serving as views) are transformed by arbitrary functions, which do not necessarily output a single "edited" view to be "putback". In this case, the link between a source and an edited view is no longer obvious, which requires a different design. We think this is a good illustration of the motto that having more is not necessarily better. The perceived advantageous machinery of having more source information available in the putback direction is not applicable; and the presumed definitional two-way bidirectional laws are unnecessary. Instead, humble right-invertibility gives us great mileage. It will be interesting to see whether other problems that appear to require full invertibility are merely right-invertible programming in disguise. One well known demand for right-invertibility is data parallel programming, where the third homomorphism theorem [Gib96, MMHT09] relies on deterministic right-inversion for generating parallel algorithms. We plan to see whether some language support will be beneficial. One known barrier that we may need to overcome is the emphasis on numerical computation in parallel programming [MMM+07], something that is not available in our current design.

Chapter 4 is another twist on conventional bidirectional programming, where edits to views are always in a sense "non-interfering". In our application of reporting the results of program analyses, the results, as annotations to ASTs, are in no way of controlling the

retrieval or putback computation; they are simply passengers on the vehicle of program transformation. In this case, a revival of a previous known technique, namely the constant complement approach, to its full generality provides us with the right tool. We have seen that with a small additional effort of maintaining the annotations from program-transformation implementers, our proposed system is able to translate analysis results in terms of internal code to ones in terms of source code, resulting in more understandable messages. The very reason for the success is the consideration of edit information: the "non-interference" property is crucial to the safety of the putback function. We believe the applicability of this usage pattern goes beyond the cases that have been discussed. For another example, consider an integrated development environment that supports matching highlighting in source code and its abstract syntax tree, based on selection of a syntax fragment in either. We can encode the text-location information as annotations, which can be edited by such a selection; and our proposed technique applies directly. It will be an interesting future direction to see how such systems can benefit from the optimization technique proposed in Chapter 5 to achieve improved run-time performance.

In Chapter 5, we take a different approach by looking not at specific problems but general principles for incremental updating of sources. Without exception, the design is based on the use of edit information; specifically, we derive an operation-based bidirectional framework that identifies the chunk(s) of a source that are not affected by a particular view editing, and only updates the part that is affected. We specify a semantic condition under which retrieval functions enjoy good locality preservation; these functions are likely to map a small edit-affected view subterm to a small update-affected source subterm. We then devise a constructive algorithm to identify the affected source subterm through indexing; and perform the update by constructing an updated subterm from the edited view, through a separate state-based putback function. Effectively, our proposal derives an operation-based bidirectional system from a state-based system, and reduces the effort of source updating from one that is proportional to the size of the data into one that is proportional to the size of the change. The resulting operation-based system is proven correct with respect to the properties of its state-based counterpart. An interesting observation from our investigation is that linear lists, widely used in functional programming, perform poorly in term of incremental updating; we had to extend our results for general trees to treat list views as special cases; and list sources will not benefit from our proposal at all, because splitting a list is a linear-time operation. This is not the only time that lists are found undesirable for performance. In parallel programming research, for exactly the same reason, cons lists are usually ruled out [Ste09, Ble10].

In the sections to follow, we will revisit some related issues and approaches in bidirectional programming, with comparison to our work in this thesis, and discuss future directions.

6.2 Totality of Putback Functions

Totality is generally a desirable property of functions, and is particularly important for bidirectional programming. While the programmer can be blamed for defining a partial function and applying it to an input that is outside of the domain, the automatic generation of putback functions in bidirectional programming shifts the responsibility of guaranteeing safety to the language designer. However, totality of putback functions is only possible with surjective retrieval functions, and is difficult to enforce – we have seen it in the designing of RINV in Chapter 3. One way to circumvent this problem is to have restricted edits to views, as we did in Chapter 4. A more general solution is to pin down the precise range of a retrieval function; and only edits resulting in views in the range are permitted – the semantic types [FCB08] in lenses are one example of this. As a matter of fact, there is already a rich body of research that allows *non-free* datatypes (datatypes that are constrained by additional specifications) to be expressed either through more advanced type [XP99, CH03, PJVWW06] or contract [Mey92, FF02, HJL06, XPJC09] systems; basing our languages on datatypes, a standard data representation, has the benefit of making them directly applicable: the specification of pre- and post- conditions of a retrieval function, can be straightforwardly reused by swapping them in the putback direction.

6.3 Combinator-Based Bidirectional Languages

The right-invertible language RINV in Chapter 3 shares a similar spirit with the approaches in [MHT04b, MHT04a, HMT04, PC10], which are based on a general purpose language, namely point-free Haskell. In particularly, RINV is very similar to the independently developed point-free lenses [PC10]: their *create*s correspond to the right inverses in RINV; and they additionally considered *put*s. Our approach is in a sense more pragmatic, as we are only interested in the part that fits our intended application, not the entire lens-framework. As discussed in Section 3.5.4, there are practical difficulties in admitting *put*s into pattern matching for ADTs.

Nevertheless, this omission of *put*s immediately rules out data-lossy retrieval functions, such as the projection functions on pairs, available in point-free lenses. We are looking at possible extension of RINV for a special kind of projection function, that only discards recoverable data. For example, consider annotating a tree with its size information at all levels in the implementation for fast traversal, and still using the bare tree as the model; the discarded size information can be regenerated from the model without the need to resort to a *put*.

6.4 Bidirectionalization

The constant complement approach used in Chapter 4 can be seen as an extension to the syntactic bidirectionalization technique in [MHN+07], with the restrictions on retrieval functions lifted. We think it is a promising direction, not because our system is strictly more powerful than the one in [MHN+07] (it is not, as we only support limited view editing), but because we believe that we found the sweet spot of balancing language expressiveness and updatability, which is made evident in the successful application of bidirectional programming to practical problems.

Chapter 5 of this thesis is originally inspired by semantic bidirectionalization [Voi09] to design a bidirectional system that is independent of the retrieval function's implementation. The conditions of regular structural recursion and aligning construction imposed on the retrieval functions for good locality preservation are purely semantic, and do not restrict the implementation in any way. In addition, our proposal also abstracts over the implementations of a state-based putback function and a view-editing function, which are used as blackboxes. This design directly contributes to the generality of our proposal, and allows most bidirectional systems to benefit from it with little adaptation. The semantic bidirectionalization brands itself as "bidirectionalization for free"; we continued this trend by offering a free operation-based bidirectionalization for a given state-based one.

Bibliography

[ABH06] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28:990–1034, November 2006.

[BC93] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[BCF+] Pablo Berdaguer, Alcino Cunha, Flávio Ferreira, Claudia Necco, José Nuno Oliveira, Hugo Pacheco, and Joost Visser. 2LT two level transformation. http://code.google.com/p/2lt/.

[BCF+10a] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 193–204, New York, NY, USA, 2010. ACM.

[BCF+10b] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.

[Ben73] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 1973.

[Ber09] Jean-Philippe Bernardy. Lazy functional incremental parsing. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 49–60, New York, NY, USA, 2009. ACM.

[BFP+08] A. Bohannon, J. Nathan Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, New York, NY, USA, January 2008. ACM.

[Bir84] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

[Ble10] Guy E. Blelloch. Functional parallel algorithms. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 247–248, New York, NY, USA, 2010. ACM.

[BMS80] Rod Burstall, Dave MacQueen, and Don Sannella. Hope: An experimental applicative language. In *Lisp and Functional Programming*, pages 136–143. ACM, 1980.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems.*, 6(4):557–575, 1981.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[Che08] James Cheney. Flux: functional updates for XML. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 3–14, New York, NY, USA, 2008. ACM.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[COV06] Alcino Cunha, José Nuno Oliveira, and Joost Visser. Type-safe two-level data transformation. In *Number 4085 in LNCS*, pages 284–289. Springer, 2006.

[CUV06] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Information and Computation*, 204:437–468, April 2006.

[DB82] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems.*, 7(3):381–416, 1982.

[DXC10] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 61–76, Berlin, Heidelberg, 2010. Springer-Verlag.

[EOW07] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*. Springer, 2007.

[EP00] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, New York, NY, USA, 2000. ACM.

[Erw96] Martin Erwig. Active patterns. In *8th Int. Workshop on Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 1996.

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Preliminary version in POPL '05.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *International Conference on Functional Programming*, pages 383–396, New York, NY, USA, 2008. ACM.

[FPZ09] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 60–74, Washington, DC, USA, 2009. IEEE Computer Society.

[GH09] Andy Gill and Graham Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19:227–251, March 2009.

[Gib96] Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[GPZ88] Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems.*, 13(4):486–524, 1988.

[Gro05] OMG: Object Management Group. MOF2.0 query/view/transformation (QVT) adopted specification. http://www.omg.org, 2005.

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009. 10.1007/s10270-008-0089-9.

[HHI+10] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 205–216, New York, NY, USA, 2010. ACM.

[HHJ11] Ralf Hinze, Thomas Harper, and Daniel W.H. James. Theory and practice of fusion. Accepted for publication in Post-proceedings of the 22nd Symposium on the Implementation and Application of Functional Languages (IFL '10), January 2011.

[HHS03] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13, New York, NY, USA, 2003. ACM.

[HJ01] Ralf Hinze and Johan Jeuring. Functional Pearl: Weaving a web. *Journal of Functional Programming*, 11(6):681–689, nov 2001.

[HJL04] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[HJL06] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *In FLOPS 06: Functional and Logic Programming: 8th International Symposium*, pages 208–225. Springer-Verlag, 2006.

[HLvI03] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71, New York, NY, USA, 2003. ACM.

[HMT04] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Partial Evaluation and Program Manipulation*, pages 178–189, New York, NY, USA, 2004. ACM.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Jay04] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6), 2004.

[JS94] Simon Peyton Jones and André Santos. Compilation by transformation in the glasgow haskell compiler, 1994.

[LG00] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, Boston, MA, USA, 2000.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages*, pages 81–92, New York, NY, USA, 2001. ACM.

[LM03] Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, London, UK, 2003. Springer.

[LP07] D. Licata and S. Peyton Jones. View patterns: lightweight views for Haskell. `http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`, 2007.

[Lut86] C. Lutz. Janus: a time-reversible language, 1986. A letter to Landauer.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[MB04] Shin-Cheng Mu and Richard Bird. Theory and applications of inverting functions as folds. *Science of Computer Programming*, 51:87–116, May 2004.

[McB01] Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

[Mee86] L. G. L. T. Meertens. Algorithmics: Towards programming as a mathematical activity. In *CWI Symposium on Mathematics and Computer Science*, number 1 in CWI-Monographs, pages 289–344. North–Holland, 1986.

[Mee98] Lambert Meertens. Designing constraint maintainers for user interaction. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3250, 1998.

[Mey92] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[MG90] Carroll Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, 1990.

[MGB04] Clare Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.

[MHN+07] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 47–58, New York, NY, USA, 2007. ACM.

[MHT04a] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2004.

[MHT04b] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 289–313. Springer, 2004.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd international joint conference on Artificial intelligence*, pages 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[MMHT09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–185, New York, NY, USA, 2009. ACM.

[MMM+07] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–155, New York, NY, USA, 2007. ACM.

[MR09] Neil Mitchell and Colin Runciman. Losing functions without gaining data: another look at defunctionalisation. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2009. ACM.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *12th Conference on Compiler Construction, Warsaw (Poland), volume 2622 of LNCS*, pages 61–76. Springer, 2003.

[NMN08] Pablo Nogueira and Juan José Moreno-Navarro. Bialgebra views: A way for polytypic programming to cohabit with data abstraction. In *Workshop on Generic Programming*, pages 61–73, New York, NY, USA, 2008. ACM.

[Oka98] Chris Okasaki. Views for Standard ML. In *ACM Workshop on ML*, 1998.

[Oli08] José N. Oliveira. Transforming data by calculation. pages 134–195, 2008.

[PC10] Hugo Pacheco and Alcino Cunha. Generic point-free lenses. In Claude Bolduc, Jules Desharnais, and Bchir Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 331–352. Springer Berlin / Heidelberg, 2010.

[PGPN96] P. Palao Gostanza, R. Peña, and M. Núñez. A new look at pattern matching in abstract data types. In *International Conference on Functional Programming*, pages 110–121, New York, NY, USA, 1996. ACM.

[PJ03] S Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[PJVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, 2006.

[Ser07] D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In Norman Ramsey, editor, *International Conference on Functional Programming*, pages 71–84. ACM Press, 2007.

[SH82] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software: Practice and Experience*, 12(11), 1982.

[SNM07] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *International Conference on Functional Programming*, pages 29–40, New York, NY, USA, 2007. ACM.

[SSW04] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91, New York, NY, USA, 2004. ACM.

[Ste07] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Model Driven Engineering Languages and Systems*, pages 1–15, 2007.

[Ste09] Guy L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *International Conference on Functional Programming*, pages 1–2, New York, NY, USA, 2009. ACM.

[Ste10] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and Systems Modeling*, 9:7–20, 2010.

[Tul00] M. Tullsen. First class patterns. In *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*. Springer, 2000.

[VHMW10] Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining syntactic and semantic bidirectionalization. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 181–192, New York, NY, USA, 2010. ACM.

[Voi09] Janis Voigtländer. Bidirectionalization for free! (Pearl). In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 165–176, New York, NY, USA, 2009. ACM.

[vSMJ10] Martijn van Steenbergen, José Pedro Magalhaes, and Johan Jeuring. Generic selections of subexpressions. In *WGP '10: Proceedings of the 6th Workshop on Generic Programming*, New York, 2010. ACM.

[Wad87a] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313, New York, NY, USA, 1987. ACM.

[Wad87b] Philip Wadler. A critique of Abelson and Sussman: Why calculating is better than scheming. *ACM SIGPLAN Notices*, 22(3):83–94, 1987.

[Wad88] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[WGM09] Hui Wu, Jeff Gray, and Marjan Mernik. Unit testing for domain-specific languages. In *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, pages 125–147, Berlin, Heidelberg, 2009. Springer-Verlag.

[WGMH10] Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. Gradual refinement: Blending pattern matching with data abstraction. In Claude Bolduc, Jules Desharnais, and Bchir Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM.

[XPJC09] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL '09: Proceedings of the 36th annual ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 41–52, New York, NY, USA, 2009. ACM.

[YAG08] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 43–54, New York, NY, USA, 2008. ACM.

[You94] S. G. Younis. *Asymptotically zero energy computing using split-level charge recovery logic*. PhD thesis, Massachusetts Inst. of Tech., Cambridge, MA., 1994.