Bidirectional Programming for Parsing and Retentive Printing

by

Zirun Zhu

Dissertation

submitted to the Department of Informatics
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI

March 2020

Abstract

Language designers usually need to implement parsers and printers for doing the conversion between program text and abstract syntax trees (ASTs), which are commonly found and constantly used in application scenarios such as resugaring, code refactoring, and observing the results of compiler optimisation. It is desirable that when producing a new piece of program text from a slightly modified (e.g. optimised) AST which corresponds to some original piece of program text, the printer can preserve the layouts, comments, and syntactic sugar contained in the original program text as much as possible. In order to do so, the common practice is to enrich ASTs with whatever information that is needed (such as comments), so that the printer can use the enriched information to produce better-printed results. However, the approach violates the design philosophy of ASTs, for ASTs should be abstract and compact and contain only essential information.

This thesis reconciles the compactness of ASTs and printing quality by proposing *retentive printing*, which takes not only an AST but also a piece of original program text, so that there is no need to enrich ASTs with unnecessary information (already in the original program text). We first propose a domain-specific language BiYacc, whose program denotes a consistent (i.e. *well-behaved*) pair of parser and retentive printer for an unambiguous context-free grammar. BiYacc is based on the theory of bidirectional transformations (BX) and in particular (state-based asymmetric) lenses, which helps to guarantee by construction that the generated parser and retentive printer pairs are always well-behaved. We show that BiYacc facilitates many tasks such as resugaring, language evolution, and simple refactoring.

However, using only unambiguous grammars can be rather inconvenient in practice, as ambiguous grammars (with disambiguation directives) are often considered more natural and human-friendly than their unambiguous versions. Therefore, we make the first move to support ambiguous grammars and tackle grammatical ambiguity in the bidirectional setting by proposing an approach based on generalised parsing and disambiguation filters, which produce all parse results and (try to) select the only desired one in the parsing direction; the filters are carefully *bidirectionalised* so

that they also work in the printing direction and will not break the well-behavedness between the pair of parser and retentive printer. We extend BiYacc with this functionality and design accessible directives for specifying production rules' associativity and (relative) priorities, which give rise to compositional and commutative filters.

Next, we explore the possibility of precise information retention and formalise the statement 'the printer can preserve the layouts, comments, and syntactic sugar contained in the original program text as much as possible'. We propose an extension of the (state-based asymmetric) lens framework, called *retentive lenses*, which satisfy a new Retentiveness law guaranteeing that if parts of an AST are unchanged, then the corresponding parts of the original program text are retained in the newly printed program text as well. We verify our idea by introducing a (new) DSL for writing tree transformations (especially between CSTs and ASTs). Retentive lenses are not state-based, and its integration into BiYacc is left to our future work.

Acknowledgements

Many people have played a crucial role in helping me pursue my PhD; without their support, all the tranquillity, mirth, and academic attainments shall have been wiped out.

I would like to express my gratitude to my (former) supervisor Zhenjiang Hu. Whilst being very busy, Zhenjiang always has a great passion for research and is willing to have a discussion with students. I remembered that there were more than ten times that our discussion started in the afternoon and lasted over twenty o'clock, during my first year here. Zhenjiang is generous to students: he supported me in attending summer schools and doing collaborative research abroad. Before I graduate, Zhenjiang suggests job opportunities as well. In the last year of my PhD, Zhenjiang moved to Peking University; consequently, there has been an adjustment to my PhD supervision system. Therefore, here, I would like to thank my current principal supervisor Ichiro Hasuo, my supervisor Hiroyuki Kato, my sub-supervisor Kanae Tsushima, and Makoto Tatsuta and Taro Sekiyama—the other five of the defence committee—and also Masato Takeichi, who showed much interest in my research and we have had several useful discussions. They spare time to listen to my presentation, read my thesis, and give advice on my research.

I am grateful to my collaborators, Hsiang-Shang Ko, Zhixuan Yang, João Saraiva, Pedro Martins, Yongzhe Zhang, and Meng Wang, for the in-depth discussions and helpful suggestions. Hsiang-Shang is a an all-rounder, a postdoctoral researcher here. Not only is he good at theorem proving, writing, and coding, but also willing to spell out complex and abstract theorems to students; he assists me from beginning to end and is like my de facto supervisor. Zhixuan is a first-year student; however, he quickly involved himself in research and helped us simplify our theoretical framework much. At the early stage in our research, João and Pedro brought up initial ideas and suggested suitable case studies; after that, João also provided guidance on parsing techniques. Yongzhe contributed to some parts of the first version of the implementation. Meng had invited me to have intensive discussions at the University of Kent and sometimes in an international conference; his suggestions were later implemented and proved to be useful. Unarguably, without them, the

research will not progress as smoothly as it had been.

I love my parents and confidants, those overseas and the ones nearby, who make my life full of comfort and happiness; we can always have a chat and share our views and feelings, freely, on whatever in the world. In particular, my mother supports me in pursuing a PhD abroad at the very beginning and offers help through the years unceasingly. (For some reasons, I would not mention their names, which will definitely constitute a rather long list, here.) Finally, I must thank the world I live in, where most areas are at peace, at least on the surface, and therefore brings good atmosphere and sufficient fund for PhD students, like me, conducting research into unimportant and unnecessary projects. Such research will never come into blossom otherwise.

Contents

1 Introduction

The same thing may have different representations. For instance, the notion of 17 can be described by 17 in Arabic numerals, by *seventeen* in English[1], and by 十七 in both Chinese and Japanese. For the general public using different languages to communicate with each other, we develop *translator*s (such as Google Translate) to do the conversion—here, the conversion for the notion of 17. As for programming languages, the situation is more or less the same: Programming languages have both *concrete syntax* specifications and *abstract syntax* specifications, and we need a pair of *parser* and *printer*[2] as the 'translator' to do the conversion between them. As Figure 1.1 shows, a piece of program text, while conforming to a concrete syntax specification, is a flat string that can be easily edited by the programmer. The parser extracts the tree structure from such a string to a concrete syntax tree (CST), and converts it to an abstract syntax tree (AST), which is a structured and simplified representation and is easier for a compiler back end to manipulate—for example, to efficiently produce code in machine languages. On the other hand, a printer converts an AST back to a piece of program text that can be understood by the user without much effort.

[1] Here we consider writing systems.

[2] People may call a printer an *unparser* or a *serialiser*. But anyway, in this thesis, we will stick to the terminology printer that is commonly used in functional programming communities.

```
/* A program to solve the
   8 queens puzzle.*/

let
  var N := 8

  type intArray = array of int

  var row := intArray [ N ] of 0
  var col := intArray [ N ] of 0
  var diag1 := ...
```

```
TLetExp [TVarDec "N" True Nothing (
    TIntExp 8),TTypeDec (TTyDec ("
    intArray",TArrayTy "int")),TVarDec "
    row" True Nothing (TArrayExp "
    intArray" (TVarExp (TSimpleVar "N"))
    (TIntExp 0)),TVarDec "col" True
    Nothing (TArrayExp "intArray" (
    TVarExp (TSimpleVar "N")) (TIntExp 0)
    ),TVarDec "diag1" True Nothing (
    TArrayExp "intArray" ...
```

Figure 1.1: Human-readable program text and its corresponding compact abstract syntax tree. (The abstract syntax tree does not contain unnecessary information such as comments and layouts.)

The conversions between program text and ASTs are commonly found and constantly used in applications such as

- resugaring (Pombrio and Krishnamurthi, 2014, 2015), where a piece of program text is parsed to its AST and evaluated (one step), and then the evaluation result is printed to new program text for the user's ease of observation;
- code refactoring (Fowler and Beck, 1999), where instead of directly modifying a piece of program text, most refactoring tools will first parse the program text into its AST, perform code refactoring on the AST, and regenerate new program text;
- observing (compiler) optimisations (Hennessy, 1982), where a piece of program text is parsed to an AST and optimised, and the user understands optimisation techniques by observing the new program text printed from the optimised AST;
- bug reporting (Traver, 2010), where a piece of program text is parsed to its AST to be checked but error messages should be displayed for the program text;.

Let us look at a concrete example (about observing optimisations) in Figure 1.2: The original program text is an arithmetic expression, containing a negation, a comment, and parentheses (one pair of which is redundant). It is first parsed to an AST (supposing that addition is left-associative) where the negation is desugared to a subtraction, parentheses are implicitly represented by the tree structure, and the comment is thrown away. Suppose that the AST is optimised by replacing `Add (Num 1) (Num 1)` with a constant `Num 2`. The user may want to observe the optimisation made by the compiler, but the AST is an internal representation not exposed to the user, so a natural

Original program text:

```
-a {- a is the variable denoting... -}
* (1 + 1 + (a))
```

Abstract syntax tree:

```
Mul (Sub (Num 0) (Var "a"))
    (Add (Add (Num 1) (Num 1)) (Var "a"))
```

Optimised abstract syntax tree:

```
Mul (Sub (Num 0) (Var "a"))
    (Add (Num 2) (Var "a"))
```

Printed result from a *conventional* printer:

```
(0 - a) * (2 + a)
```

Printed result from our *retentive* printer:

```
-a {- a is the variable denoting... -}
* (2 + (a))
```

Figure 1.2: Comparison between conventional printing and retentive printing. (The abstract syntax tree is represented in Haskell's syntax. The reader unfamiliar with Haskell may view the AST in this way: the root of the AST is a node named `Mul`, which has two subtrees named `Sub` and `Add` respectively.)

idea is to propagate the change on the AST back to the program text to make it easy for the user to check where the changes are. With a conventional printer, however, the printed result will probably mislead the user into thinking that in addition to replacing `1 + 1` with `2`, the negation `-a` is also replaced by a subtraction `(0 - a)` by the compiler; also, the loss of comment makes it harder for the user to compare the updated and original versions of the text. Obviously, this is because the AST does not contain enough information such as the comments and constructs for the syntactic sugar negation (in Figure 1.2).

To make the printed results better, existing approaches choose to enrich ASTs with whatever information that is needed, such as comments and layouts for code refactoring, tags (marking from which CST construct an AST construct is parsed) for resugaring, and line and column numbers for bug reporting. The choice of enriching ASTs indeed leads to a problem, as it violates the design philosophy of ASTs—ASTs should be abstract, compact, and contain only essential information. Moreover, this is far from economical, because the compiler of a language has already defined a (relatively clean) AST, but each application still defines its own version.

This thesis reconciles the compactness of ASTs and printing quality by proposing *retentive* printing. Different from a conventional printer, a retentive printer takes a piece of program text and an AST that is usually slightly modified from the AST corresponding to the original program text; it produces a new piece of program text in a way that not only considers all the modification to the AST but also tries to preserve the information in the original program text (that is not in the AST). In this way, the syntactic sugar, comments, and layouts in the unmodified parts of the program text can still be preserved (even using abstract and compact ASTs), which can be seen clearly from the result of using our retentive printer on the above arithmetic expression example in Figure 1.2. It is worth noting that retentive printing is a generalisation of the conventional notion of printing, because a retentive printer can accept an AST and an empty piece of program text, in which case it will behave just like a conventional printer, producing a new piece of program text depending on the AST only.

Furthermore, to ease the work of developing compiler front ends, we design domain-specific languages (DSLs) in which a single specification compiles to a pair of parser and retentive printer that are *consistent* with each other—for instance, the program text printed from an AST should be parsed to the same tree. The unification is indeed helpful, for it overcomes two drawbacks of designing a parser and a printer separately for a language: (i) It takes extra effort to guarantee that the two closely related components are consistent with each other. (ii) When the language evolves, we need to revise *both* components and prove their consistency again. The idea of generating a pair of parser and conventional printer using DSLs is not new (Boulton, 1996; Brabrand et al., 2008;

Duregård and Jansson, 2011; Matsuda and Wang, 2018b; Rendel and Ostermann, 2010; van den Brand et al., 2001), however, we are yet the first to consider a consistent pair of parser and retentive printer.

The proposed solutions in this thesis are inspired by the research on *bidirectional transformations* (BX) (Abou-Saleh et al., 2018; Czarnecki et al., 2009) and in particular *asymmetric lenses* (Foster, 2009; Foster et al., 2007). We will explain, in Chapter 2, that a pair of parser and retentive printer indeed form an asymmetric lens that *synchronise* program text and ASTs, and in this way, the consistency between a pair of parser and retentive printer is naturally modelled as *well-behavedness* of a lens. To summarise, our solution

- reconcile the compactness of ASTs and printing quality, and
- allows the user to design consistent parser and (retentive) printer pairs for a fully disambiguated context-free grammar (CFG) in a single specification.

1.1 Organisation and Contributions

We organise the remainder of the thesis as follows. Chapter 3–Chapter 5 are the main contributions of this thesis, which include both the ideas and foundations, and the language design and implementation.

Chapter 2. We introduce bidirectional transformations, (asymmetric) lenses, and our method of modelling parsing and printing as state-based asymmetric lenses. In practice, since it is hard to synchronise unstructured program text and structured ASTs, we further divide the synchronisation between program and ASTs into two: an isomorphism between program text and CSTs, and a lens between CSTs and ASTs. We present some related work which justifies the need for yet another way to unify parsing and printing and the method we adopt.

Chapter 3. We present a concrete solution to the unification of parsing and retentive printing: We deliberate on the language design of the basic version of a DSL, BiYacc, which can generate a consistent pair of parser and retentive printer as a lens from a single specification for unambiguous grammars. This chapter is adapted from our early papers on BiYacc (Zhu et al., 2015, 2016).

Chapter 4. We make an extension to our solution (mainly) for the isomorphism part (between program text and CSTs). In Chapter 3, we state that our solution only handles unambiguous

grammars, which will be rather inconvenient in practice. Now, we explore the possibility of supporting ambiguous grammars with *disambiguation directives* while still keeping the well-behavedness (i.e. consistency) between a generated parser and printer pair. We achieve this by proposing *bidirectionalised filters* (bi-filters) based on the technology of *generalised parsing* and (unidirectional) filters and integrating them into BiYacc. This chapter is revised from our journal paper on the extended BiYacc (Zhu et al., 2020a).

Chapter 5. We make another extension to our solution mainly for the lens part (between CSTs and ASTs). Over the years, we find that while lenses are designed to retain information—for instance, information such as syntactic sugar and comments in the updated program text—well-behavedness (as the consistency) says very little about the retention of information. To guarantee the retention of information theoretically, we propose Retentiveness and an extension of the original lenses, called *retentive lenses*. We verify our idea by introducing a (new) DSL for writing tree transformations (especially between CSTs and ASTs). Retentive lenses are not state-based, and its integration into BiYacc is left to our future work. This chapter is recast from our unpublished paper on retentive lenses (Zhu et al., 2020b).

Chapter 6. We conclude the thesis by reviewing our contributions and discussing some possible future work.

An online tool that implements the approach described in the thesis can be accessed at https://biyacc.k331.one. We assume basic knowledge about functional programming languages and their notations, in particular Haskell (Bird, 2014; Marlow et al., 2010). In Haskell, an argument of function application does not need to be enclosed in (round) parentheses, i.e. we write $f\ x$ instead of $f\ (x)$; type variables are implicitly universally quantified, i.e. $f :: a \to b \to a$ is the same as $f :: \forall a\ b.\ a \to b \to a$ where :: means *has type*. Additionally, we omit universal quantification for free variables in an equation; for instance, $id\ x = x$ is in fact $\forall x.\ id\ x = x$. Throughout the thesis, we typeset general definitions and properties in *math style* and specific examples in `code style`.

2 Parsing and Printing as Lenses

This chapter provides foundations of bidirectional programming for parsing and printing. We commence with an introduction to bidirectional transformations (Section 2.1) by presenting the origin and a particular framework called *state-based asymmetric lenses*. Then we model a consistent pair of *parse* and *print* functions as a state-based asymmetric lens (Section 2.2). Finally, we present general related work as the rationale of our approach (Section 2.3). The presentation of more in-depth related work assumes familiarity with the technical content of this thesis, and is thus given at the end of each chapter (Sections 3.4, 4.7, and 5.5).

2.1 Bidirectional Transformations and Lenses

The theoretical foundation of the thesis is bidirectional transformations (BX) (Abou-Saleh et al., 2018; Czarnecki et al., 2009), in particular (asymmetric) lenses (Foster, 2009; Foster et al., 2007), which are believed to be a satisfactory solution (Foster et al., 2007) to the long-standing *view update* problem (Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982; Gottlob et al., 1988) in the (relational) database community since 1970s. The view update problem can be better explained with the help of Figure 2.1: We call a database s that contains comprehensive data a *source*, from

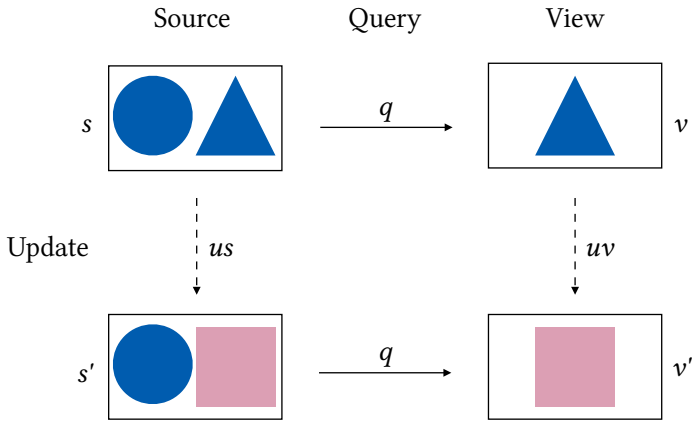


Figure 2.1: The view update problem.

which we can use a *query* q to obtain data of our interest; the result of the query q on the source s forms a *view* v, which in general contains only a portion of the data in the source. For the sake of convenience or security, the user is sometimes supposed to modify the view v instead of the original source s. Given a view v' modified from v by an update uv, the view update problem is how to calculate an update us on the source s that transforms s to s' such that $q\ s' == v'$.

While the (relational) database community has studied the view update problem more from a theoretical aspect and proposed many solutions, the bidirectional transformation community tackles the problem more from a language-oriented perspective, that is, to provide elegant (domain-specific) programming languages which support the user in writing a single program that can be interpreted both as a query and as an update. In this way, the query and update are guaranteed to work in harmony and the single program is also easy to maintain (Abou-Saleh et al., 2018). Although there are many bidirectional programming frameworks (Section 2.3.2), in this thesis, we will mainly introduce the *state-based asymmetric lens framework* (Foster, 2009; Foster et al., 2007) since other frameworks are marginally relevant to the particular approach we adopt to building pairs of consistent parsers and printers. The state-based asymmetric lens framework is illustrated in Figure 2.2, where a lens consists of a pair of *get* and *put* functions. While the *get* function is the same as *query*, the *put* function does not calculate an update us from uv; instead, *put* accepts a source s and a (modified) view v' and directly propagate the changes in v' back to s. That is, we abstract out the notion of update (i.e. uv and us) and use the states of data (i.e. s and v') only. State-based lenses have become popular since the pioneering work of Foster et al. (2007) on a combinatorial language for bidirectional tree transformations.

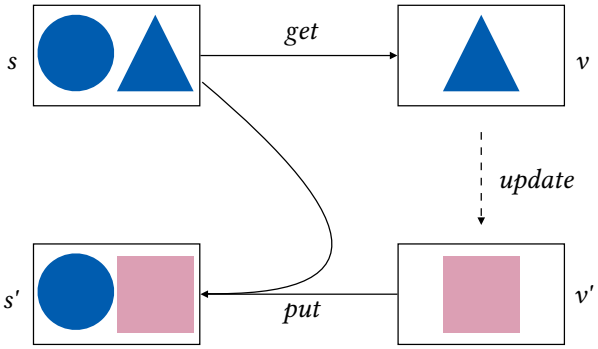


Figure 2.2: The framework of state-based asymmetric lenses.

Definition 2.1.1 (Lenses)**.** A lens between a source type S and a view type V is a pair of functions

$$get :: S \to V$$
$$put :: S \times V \to S$$

satisfying the well-behavedness laws (Stevens, 2008):

$$put\ (s, v) = s' \quad \Rightarrow \quad get\ s' = v \qquad \text{(Correctness)}$$
$$get\ s = v \quad \Rightarrow \quad put\ (s, v) = s \qquad \text{(Hippocraticness)}$$

Intuitively, a *get* function extracts a part of a source of interest to the user as a view, and a *put* function takes a source and a view and produces an updated source incorporating information from the view. The Correctness law enforces that *put* must embed all information of the view into the updated source, so the view can be recovered from the source by *get*, while the Hippocraticness law prohibits *put* from performing unnecessary updates by requiring that putting back a view directly extracted from a source by *get* must produce the same, unmodified source.

2.2 Parsing and Printing as Lenses

When we say a pair of (conventional) parser and printer are *consistent* with each other, we want to ensure the following two properties:

$$print\ t = s \quad \Rightarrow \quad parse\ s = t$$

$$parse\ s = t \quad \Rightarrow \quad print\ t = s$$

The first property says that a piece of program text s printed from an abstract syntax tree t should be parsed to the same tree t; the second property asserts that updating a piece of program text s with an abstract syntax tree t parsed from s should leave s unmodified, including syntactic sugar and formatting details such as parentheses and whitespace.

But this seems impossible, for the abstract syntax tree t usually does not (should not) include syntactic sugar and formatting details—which, however, can be found in the program text s. Thus, a reasonable idea is to let *print* also take the program text s as input; then we have the following (inverse-like) consistency properties for a pair of parser and retentive printer:

$$\begin{aligned} print\ (s, t) = s' \quad &\Rightarrow \quad parse\ s' = t \\ parse\ s = t \quad &\Rightarrow \quad print\ (s, t) = s \end{aligned}$$

Observant readers might have noticed that the two (inverse-like) consistency properties are exactly Correctness and Hippocraticness of asymmetric lenses, if we regard *parse* as *get* and *print* as *put*. Thus, a *consistent* pair of parser and retentive printer indeed forms a well-behaved lens.

However, in practice, directly synchronising unstructured program text and structured ASTs turns out to be a difficult task. Fortunately, we can decompose the lens between program text and ASTs into the composition of a (partial) isomorphism between program text and CSTs and a lens between CSTs and ASTs instead. This is possible because the production rules in a context-free grammar dictate how to produce (valid) program text from nonterminals, and a CST can be regarded as encoding one particular way of producing program text using the production rules. Therefore we can always first establish an isomorphism between unstructured program text and its unique CST[1], and later synchronise the structured CST and AST using a lens, as shown in Figure 2.3. Moreover, as the *parse* and *print* semantics are potentially *partial* in the real world, we also need to take partiality into account when choosing a BX framework in which to model *parse* and *print*.

2.2.1 Composition of Isomorphisms and Lenses

We start from the definition of (partial) isomorphisms.

[1] If the grammar is unambiguous, the isomorphism is also unique. We will expound more on this in Section 3.2.2.

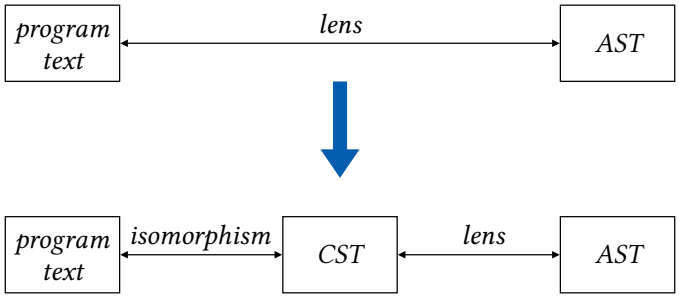


Figure 2.3: Decomposing a lens into the composition of an isomorphism and another lens.

Definition 2.2.1 (Isomorphism)**.** A (partial) *isomorphism* between two types A and B is a pair of functions

$$\begin{aligned} to &:: A \to \text{Maybe } B \\ from &:: B \to \text{Maybe } A \end{aligned}$$

such that

$$to\ a = \text{Just } b \quad \Leftrightarrow \quad from\ b = \text{Just } a\ .$$

Partiality is explicitly represented by making the functions return Maybe-values: a *to* or *from* function returns Just r where r is the result, or Nothing if the input is not in the domain. We will show that the transformations between program text and CSTs exactly form a partial isomorphism in Section 3.2.2.

Now we adapt the definition of lenses (Definition 2.1.1) to its partial version (Macedo et al., 2013; Pacheco et al., 2014a).

Definition 2.2.2 (The Partial Version of Lenses)**.** A lens between a source type S and a view type V is a pair of functions

$$\begin{aligned} get &:: S \to \text{Maybe } V \\ put &:: S \times V \to \text{Maybe } S \end{aligned}$$

satisfying the partial version of well-behavedness laws:

$$put\ (s, v) = \text{Just } s' \quad \Rightarrow \quad get\ s' = \text{Just } v \qquad \text{(Correctness)}$$

$$get\ s = \text{Just } v \quad \Rightarrow \quad put\ (s, v) = \text{Just } s \qquad \text{(Hippocraticness)}$$

The *parse* and *print* semantics will be the pair of functions *get* and *put* in a (partial) lens, required by definition to satisfy the two well-behavedness laws, which are exactly the consistency properties reformulated in a partial setting:

Definition 2.2.3 (The Partial Version of Consistency Properties)**.**

$$\begin{aligned} \mathit{print}\ (s, t) = \mathsf{Just}\ s' \quad &\Rightarrow \quad \mathit{parse}\ s' = \mathsf{Just}\ t \\ \mathit{parse}\ s = \mathsf{Just}\ t \quad &\Rightarrow \quad \mathit{print}\ (s, t) = \mathsf{Just}\ s \end{aligned}$$

Definition 2.2.4 (Composition of Isomorphism and Lenses)**.** Given an isomorphism (*to* and *from*) between A and B and a lens (*get* and *put*) between B and C, we can compose them to form a new lens between A and C, whose components *get*′ and *put*′ are defined by

$$\begin{aligned} \mathit{get}' &:: A \to \mathsf{Maybe}\ C \\ \mathit{get}'\ a &= \mathit{to}\ a \gg\!= \mathit{get} \\ \\ \mathit{put}' &:: A \times C \to \mathsf{Maybe}\ A \\ \mathit{put}'\ (a, c) &= \mathit{to}\ a \gg\!= \lambda\ b \to \mathit{put}\ (b, c) \gg\!= \mathit{from} \end{aligned}$$

where

$$\begin{aligned} (\gg\!=) &:: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\ \mathsf{Just}\ x \gg\!= f &= f\ x \\ \mathsf{Nothing} \gg\!= f &= \mathsf{Nothing}\ . \end{aligned}$$

This is specialised from the standard definition of lens composition (Foster et al., 2007)—an isomorphism can be lifted to a lens (whose *put* component passes its second argument to the *from* component of the isomorphism), which can then be composed with another lens to give rise to a new lens. We thus have the following lemma.

Lemma 2.2.5. Any lens resulted from the composition in Definition 2.2.4 is well-behaved.

2.3 Rationale of Our Approach

Before implementing our solution concretely (in the next section), here we introduce other approaches to the unification of parsing and printing and explain why we need yet another one (Section 2.3.1); we present several other bidirectional transformation frameworks and briefly discuss the reason of our choice (Section 2.3.2); we also compare two approaches to designing

state-based lenses, namely *get-based* approaches and *put-based* approaches, and examine why put-based approaches are more suitable for the unification of parsing and printing (Section 2.3.3).

2.3.1 Other Approaches to the Unification

Much research has been devoted to describing parsers and printers in a single program (Boulton, 1996; Duregård and Jansson, 2011; Matsuda and Wang, 2018b; Rendel and Ostermann, 2010; van den Brand et al., 2001). Despite their advantages, these domain-specific languages are not designed for (and therefore cannot deal with) synchronisation between program text and ASTs: their generated printers are *pretty printers*, which try to produce good-looking program text that is readable for programmers. To be precise, van den Brand et al., Rendel and Ostermann, and Duregård and Jansson use pre-defined pretty printing strategies which cannot be customised by the user; on the other hand, Boulton and Matsuda and Wang provide the functionality of smartly adjusting the layout in the printed program text according to the user's specification. We believe that the difference mainly lies in the fact that van den Brand et al., Rendel and Ostermann, and Duregård and Jansson let the user describe either parsers or grammars while Boulton and Matsuda and Wang let the user specify printers.

Regarding the properties to be satisfied by a parser and printer pair (Definition 2.2.3), Boulton, van den Brand et al., and Duregård and Jansson do not mention any, just like most industrial tools such as Xtext (Friese et al., 2008); in contrast, both Rendel and Ostermann and Matsuda and Wang propose properties similar to ours. Rendel and Ostermann uses a *partial isomorphism* (Definition 2.2.1) to construct a consistent pair of parser and printer; Matsuda and Wang use grammar inversion to produce a consistent parser from the printer specification; the parser and printer pair satisfies a non-deterministic version of the inverse properties

$$\mathit{print}\ t \Downarrow_{ND} s \Leftrightarrow t \in \mathit{parse}\ s\ .$$

This *print* function can produce either pretty or ugly strings, non-deterministically. Nevertheless, both Rendel and Ostermann's and Matsuda and Wang's properties 'tend to' fail when a language has syntactic sugar: As for Rendel and Ostermann's approach, in fact, there is no hope of establishing such partial isomorphism if the language has syntactic sugar, for the *parse* function is not injective in this case; for instance, *parse* may convert both a negation and a subtraction to the same AST[1]. As for Matsuda and Wang's approach, the property will be correct if the non-determinism takes

[1]Unless we let the *parse* function fail when the input is negation (or subtraction). But this is not what we want.

the syntactic sugar into account, but we did not find the consideration either in (the explanation of the non-determinism in) their paper or in their implementation.

To be short, the essential factor distinguishing our approach from others is that our printer is *retentive* and can deal with synchronisation, while all the above-mentioned existing approaches are targeted at producing good-looking program text and hence cannot handle synchronisation.

2.3.2 Other Bidirectional Transformation Frameworks

We only introduced (asymmetric) lenses because they are considered an elegant solution to the view update problem (Foster et al., 2007)—where views are ASTs and sources are program text in our setting. Here, we expound more on bidirectional transformations.

Most of the bidirectional transformation frameworks can be divided into three categories in terms of the types of the *get* and *put* functions (Pacheco, 2012):

- *mappings*, in which $get :: S \to V$ and $put :: V \to S$ form a bijection and are inverse functions to each other (due to the degenerate well-behavedness);
- *asymmetric lenses* or simply *lenses*, in which $get :: S \to V$ and $put :: S \times V \to S$ are asymmetric and satisfy well-behavedness, as introduced in Definition 2.1.1.
- *symmetric lenses* or *maintainers*, in which $get :: S \times V \to V$ and $put :: V \times S \to S$ are symmetric and satisfy the symmetric version of well-behavedness.

Under each framework, there are many different approaches and we can classify them as *set-based* (or *state-based*), *edit-based*, and *delta-based* ones (Johnson et al., 2016) from the perspective of *update representation*: In set-based approaches, an update on a source or view is simply represented by its (post-) state; in edit-based approaches, an update is represented by a sequence of operations, such as move, delete, and insert, that are finally applied to a pre-state to produce a post-state; in delta-based approaches, an update is represented by a delta between a pre-state and a post-state—the notion of delta is a little abstract and we will not go further into the detail here; but concretely, deltas can also (but not necessarily) be implemented as edit operations (Diskin et al., 2011a). Despite the difference, the three approaches are closely related and their relations are summarised by Johnson et al. (2016).

In general, set-based approaches are the most flexible ones (Pacheco, 2012) as they demand only the states of the sources and views; no more precise information regarding the update operation is required. However, they may not produce desired results for many applications. By

contrast, the edit-based and delta-based approaches are usually tightly coupled with applications and hence less flexible; but they are likely to use the additional information to produce better update results (Foster, 2009; Pacheco, 2012). We opt for state-based lenses because in our setting, parsers and printers usually take only the states of program text and ASTs; moreover, a parser and printer pair of a language may be used in many language-related applications such as bug reporting, code refactoring, and resugaring mentioned in the introduction, so that we need an approach that is flexible and loosely coupled with any application.

2.3.3 Get-based and Put-based Lens Languages

Usually, lens languages reduce the burden of the user by enabling the user to think in either the *get* direction or the *put* direction: When using lens languages to write transformations, the user describes the semantics of the two transformations by merely considering the *get* or *put* direction; the other is 'automatically derived' (fixed) by the pre-defined semantics of the lenses and is well-behaved by design. A lens language is called *get-based* if it encourages the user to describe the *get* behaviour and is called *put-based* if it encourages the user to specify the *put* behaviour (Ko and Hu, 2018). Typical state-based lenses are tree lenses (Foster et al., 2007), Boomerang (Bohannon et al., 2008), quotient lenses (Foster et al., 2008), matching lenses (Barbosa et al., 2010), BiFluX (Pacheco et al., 2014b), and BiGUL (Ko et al., 2016); among them, tree lenses, Boomerang, quotient lenses, and matching lenses are get-based, while BiFluX and BiGUL are put-based.

In the situation of parsing and (retentive) printing, we prefer put-based design to get-based design[1]. This can be easily justified by the concrete example which we previously mentioned in Figure 1.2. Supposing that the AST is `Sub (Num 0) (Var "a")` and the program text is missing, by specifying printing strategies (i.e. describing the *put* behaviour), the user is able to declare that s/he wants either `0 - a` or `-a`, which does not affect the parsing behaviour (*get* behaviour) because both `0 - a` and `-a` will be parsed to `Sub (Num 0) (Var "a")`. That is, the user is granted the control over generating desired program text—but s/he still does not need to consider the parsing direction. By contrast, since get-based frameworks by design should not let the user consider the semantics of the printing direction, the user does not (and should not) have the control over the printed program text. How the programmer can effectively work with put-based paradigm has been more formally explained in terms of a Hoare-style logic, using the BiGUL language as an example (Ko and Hu, 2018).

[1] Whilst we prefer put-based design, it still cannot meet the requirement of precise information retention in complex situations. This will be explained clearly in Chapter 5 and especially in that chapter's related work.

The difference is indeed reasonable, as there is a lemma (Foster, 2009) indicating that '*put* is the essence of bidirectional programming' (Fischer et al., 2015).

Lemma 2.3.1 (*put* determines *get*)**.** Given a *put* function, there is at most one *get* function that forms a (well-behaved) lens with this *put* function (Foster, 2009).

Thanks to this, in this thesis we can focus on the printing (*put*) behaviour, leaving the parsing (*get*) behaviour implicitly but unambiguously specified.

3

A DSL for Simultaneously Specifying Consistent Parser and Printer Pairs

In the last chapter, we gave an introduction to the foundation for modelling a consistent pair of *parse* and *print* functions as a well-behaved lens. Now, it is time to present a concrete implementation. In this chapter, we focus on the language design of the basic version of BiYacc—a domain-specific language which enables the user to design consistent parser and (retentive) printer pairs for unambiguous grammars[1] simultaneously—with the help of the running example about arithmetic expressions shown in Figure 1.2. We start with an overview of BiYacc (Section 3.1), which gives the reader an impression of what a program in BiYacc looks like and how the program executes. Then we expound on the design details of the DSL (Section 3.2). Related work is presented in the last section of this chapter, where we compare BiYacc with several other similar systems.

[1] We will support ambiguous grammars in Chapter 4.

```
1  #Abstract
2  data Arith = Num Int
3             | Var String
4             | Add Arith Arith
5             | Sub Arith Arith
6             | Mul Arith Arith
7             | Div Arith Arith
8
9  #Concrete
10 Expr   -> Expr '+' Term
11         | Expr '-' Term
12         | Term ;
13
14 Term   -> Term '*' Factor
15         | Term '/' Factor
16         | Factor ;
17
18 Factor -> '-' Factor
19         | Numeric
20         | Identifier
21         | '(' Expr ')' ;
22 #Directives
23 LineComment:  "//"       ;
24 BlockComment: "/*"  "*/" ;
25
26 #Actions
27 Arith +> Expr
28   Add x y  +>  [x +> Expr] '+' [y +> Term];
29   Sub x y  +>  [x +> Expr] '-' [y +> Term];
30   e        +>  [e +> Term];
31 ;;
32 Arith +> Term
33   Mul x y  +>  [x +> Term] '*' [y +> Factor];
34   Div x y  +>  [x +> Term] '/' [y +> Factor];
35   e        +>  [e +> Factor];
36 ;;
37 Arith +> Factor
38   Sub (Num 0) y  +>  '-' [y +> Factor];
39   Num i          +>  [i +> Numeric];
40   Var n          +>  [n +> Identifier];
41   e              +>  '(' [e +> Expr] ')';
42 ;;
```

Figure 3.1: A BiYacc program for the arithmetic expression example.

3.1 A First Look at BiYacc

We first give an overview of BiYacc by going through the BiYacc program shown in Figure 3.1 that deals with the arithmetic expression example in Figure 1.2. The program consists of definitions of the abstract syntax, concrete syntax, directives, and actions for retentively printing ASTs to CSTs; we will introduce them in order.

3.1.1 Syntax Definitions

Abstract Syntax. The abstract syntax part, which starts with the keyword `#Abstract`, is just one or more definitions of Haskell data types. In our example, the abstract syntax is defined in lines 2–7 by a single datatype `Arith` whose elements are constructed from constants and arithmetic operators. Different constructors—namely `Num`, `Var`, `Add`, `Sub`, `Mul`, and `Div`—are used to construct different kinds of expressions.

Concrete Syntax. The concrete syntax part, beginning with the keyword `#Concrete`, is defined by a context-free grammar. For our expression example, in lines 10–21 we use a standard unambiguous grammatical structure to encode operator precedence and order of association, involving

three nonterminal symbols `Expr`, `Term`, and `Factor`: An `Expr` can produce a left-sided tree of `Terms`, each of which can in turn produce a left-sided tree of `Factors`. To produce right-sided trees or operators of lower precedence under those with higher precedence, the only way is to reach for the last production rule `Factor -> '(' Expr ')'`, resulting in parentheses in the produced program text. There are also predefined nonterminals *Numeric* and *Identifier*, which produce numerals and identifiers respectively.

Directives. The `#Directives` part defines the syntax of comments and disambiguation directives. For example, line 23 shows that the syntax for single line comments is "`//`"[1], while line 24 states that "`/*`" and "`*/`" are respectively the beginning mark and ending mark for block comments. Since the grammar for arithmetic expressions is unambiguous, there is no need to give any disambiguation directive for this example (whereas the ambiguous version of the grammar in Figure 4.1 needs to be augmented with a few such directives).

3.1.2 Printing Actions

The main part of a BiYacc program starts with the keyword `#Actions` and describes how to update a CST with an AST. For our expression example, the actions are defined in lines 27–42 in Figure 3.1. Before explaining the actions, we should first say that program text is identified with CSTs when programming BiYacc actions: Conceptually, whenever we write a piece of program text, we are actually describing a CST rather than just a sequence of characters, which has been mentioned in the paragraph before Section 2.2.1. We will expound on this identification of program text with CSTs in Section 3.2.2.3 in detail.

The `#Actions` part consists of groups of actions, and each group begins with a 'type declaration' of the form *HsType* '+>' *Nonterminal* stating that the actions in this group specify updates on CSTs generated from *Nonterminal* using ASTs of type *HsType*. Informally, given an AST and a CST, the semantics of an action is to perform pattern matching simultaneously on both trees, and then use components of the AST to update corresponding parts of the CST, possibly recursively. (The syntax '+>' suggests that information from the left-hand side is embedded into the right-hand side.) Usually the nonterminals in a right-hand side pattern are overlaid with update instructions, which are also denoted by '+>'.

[1] While single quotation marks are for characters, double quotation marks are for strings. For simplicity, the user can always use double quotation marks.

Let us look at a specific action—the first one for the expression example, at line 28 of Figure 3.1:

```
Add x y +> [x +> Expr] '+' [y +> Term];
```

The AST-side pattern `Add x y` is just a Haskell pattern; as for the CST-side pattern, the main intention is to refer to the production rule `Expr -> Expr '+' Term` and use it to match those CSTs produced by this rule—since the action belongs to the group `Arith +> Expr`, the part '`Expr ->`' of the production rule can be inferred and thus is not included in the CST-side pattern. Finally we overlay '`x +>`' and '`y +>`' on the nonterminal symbols `Expr` and `Term` to indicate that, after the simultaneous pattern matching succeeds, the subtrees `x` and `y` of the AST are respectively used to update the left and right subtrees of the CST.

Having explained what an action means, we can now explain the semantics of the entire program. Given an AST and a CST as input, first a group (of actions) is chosen according to the types of the trees. Then the actions in the group are tried in order, from top to bottom, by performing simultaneous pattern matching on both trees. If pattern matching for an action succeeds, the updating operations specified by the action is executed; otherwise the next action is tried. Execution of the program ends when the matched action specifies either no updating operations or only updates to primitive data types such as `Numeric`. BiYacc's most interesting behaviour shows up when all actions in the chosen group fail to match—in this case a suitable CST will be created. The specific approach adopted by BiYacc is to perform pattern matching on the AST only and choose the first matched action. A suitable CST conforming to the CST-side pattern is then created, and after that the whole group of actions is tried again. This time the pattern matching will succeed at the action used to create the CST, and the program will be able to make further progress. For instance, assuming that the source is `1 * 2` while the view is `Add (Num 1) (Num 2)`, a new source skeleton representing `_ + _` will be created and the `_` part will be updated recursively later. We will elaborate more on this in Section 3.2.

Deep Patterns. By using deep patterns, we can write actions that establish nontrivial relationships between CSTs and ASTs. For example, the action at line 38 of Figure 3.1 associates abstract subtraction expressions whose left operand is zero with concrete negated expressions; this action is the key to preserving negated expressions in the CST. For an example of a more complex CST-side pattern: Suppose that we want to write a pattern that matches those CSTs produced by the rule `Factor -> '-' Factor`, where the inner nonterminal `Factor` produces a further `'-' Factor` using the same rule. This pattern is written by overlaying the production rule

on the first nonterminal `Factor` (an additional pair of parentheses is required for the expanded nonterminal): `'-' (Factor -> '-' Factor)`. More examples involving this kind of deep patterns can be found in Section 3.3.

Layout and Comment Preservation. The retentive printer generated by BiYacc is capable of preserving layouts and comments, but, perhaps mysteriously, in Figure 3.1 there is no clue as to how layouts and comments are preserved. This is because we decide to hide layout preservation from the user, so that the more important logic of abstract and concrete syntax synchronisation is not cluttered with layout preserving instructions. Our approach is fairly simplistic: We store layout information following each terminal in an additional field in the CST implicitly, and treat comments in the same way as layouts. During the printing stage, if the pattern matching on an action succeeds, the layouts and comments after the terminals shown in the right-hand side of that action are preserved; on the other hand, layouts and comments are dropped when a CST is created in the situation where pattern matching fails for all actions in a group. The layouts and comments before the first terminal are always kept during the printing process.

Parsing Semantics. So far we have been describing the retentive printing semantics of the BiYacc program, but we may also work out its parsing semantics intuitively by interpreting the actions from right to left, converting the production rules to the corresponding constructors. (This might remind the reader of the usual Yacc (Johnson, 1975) actions.) In fact, this thesis will not define the parsing semantics formally, because the parsing semantics is completely determined by the retentive printing semantics (Lemma 2.3.1): If the actions are written with the intention of establishing some relation between the CSTs and ASTs, then BiYacc will be able to derive the only well-behaved parser, which respects that relation.

3.1.3 Program Execution

To get a solid feeling of how BiYacc works, let us go through the execution of the program in Figure 3.1 on the program text and optimised AST shown in Figure 1.2 to see how the updated program text is produced.

At the beginning, the types of the input CST and AST are assumed to match those of the first group of actions, and we will try the actions in the first group in order. Consider the first action at line 28: since the input AST is a multiplication (`Mul ...`) which does not match the AST-side pattern `Add x y` and the CST (`-a * ...`) is produced from `Expr -> Term` (followed by

`Term -> Term '*' Factor`) that does not match the CST-side pattern `Expr '+' Term`, both the pattern matching on the AST-side and the CST-side fail. So, we need to try other actions in order: We fail at line 29 but succeed at line 30, because the AST-side pattern `e` can match any input AST and the input CST indeed is produced from `Expr -> Term`. This action tells us to update the CST's subtree produced from a `Term` with the whole AST; since now the type of the AST is still `Arith` but the type of the (subtree of the) CST to be updated becomes `Term`, we move on to the second group `Arith +> Term`. In the second action group, the action at line 33 matches. This action tells us to further update the subtree `-a` with `Sub (Num 0) (Var "a")` and the and the subtree `(1 + 1 + (a))` with `Add (Num 2) (Var "a")` respectively. Note that at this point, the comments and layout information after the terminal `*` is preserved[1].

For the update to be performed on the left-hand side of the multiplication, i.e. using `Sub (Num 0) (Var "a")` to update `-a`, the second action group `Arith +> Term` is chosen because the update strategy `[x +> Term]` indicates that we should use `x` (of type `Arith`, inferred from the AST-side pattern) to update the (subtree of the) CST produced from a `Term`. This time, the action at line 35 matches, which tells us to proceed with the third action group `Arith +> Factor`. Then we reach the action at line 38, where we use `Sub (Num 0) (Var "a")` to update `-a` and preserve the negation; finally we reach line 40, which uses `Var "a"` in the AST to update the variable `a` in the CST and preserves the comments and layout information after it. (The update is still performed even though the concrete and abstract sides are the same.)

The update to be performed on the right-hand side of the multiplication is executed similarly: The third action group is selected and the action at line 41 matches; now, the outermost parentheses of `(1 + 1 + (a))` are preserved and we are to update the subtree `1 + 1 + (a)` produced from the `Expr` inside the parentheses with the AST `Add (Num 2) (Var "a")`. This time, the action at line 28 matches and `1 + 1` will be updated with `Num 2` and `(a)` will be updated with `Var "a"` respectively. When we use `Num 2` to update `1 + 1`, none of the actions of the first group matches; in this case, we need to produce a new CST from scratch according to the input AST. So, we find the first action, at line 30, whose AST-side pattern `e` matches the input AST, and create a new CST according to the action's CST-side pattern `Expr -> Term`; this leads us to go into the second group. The remaining execution sequences are as follows: Actions at lines 33, 34, and 35 fail to match the CST and AST simultaneously, so the CST is reshaped to a `Factor` according to the AST at line 35 and we run into the third action group. Actions at lines 38, 39, 40, and 41 fail to match the CST and AST simultaneously, so the CST is adapted to a `Numeric` value at line 39. The newly created numeric

[1] In this example, however, there is no comment after the terminal `*`

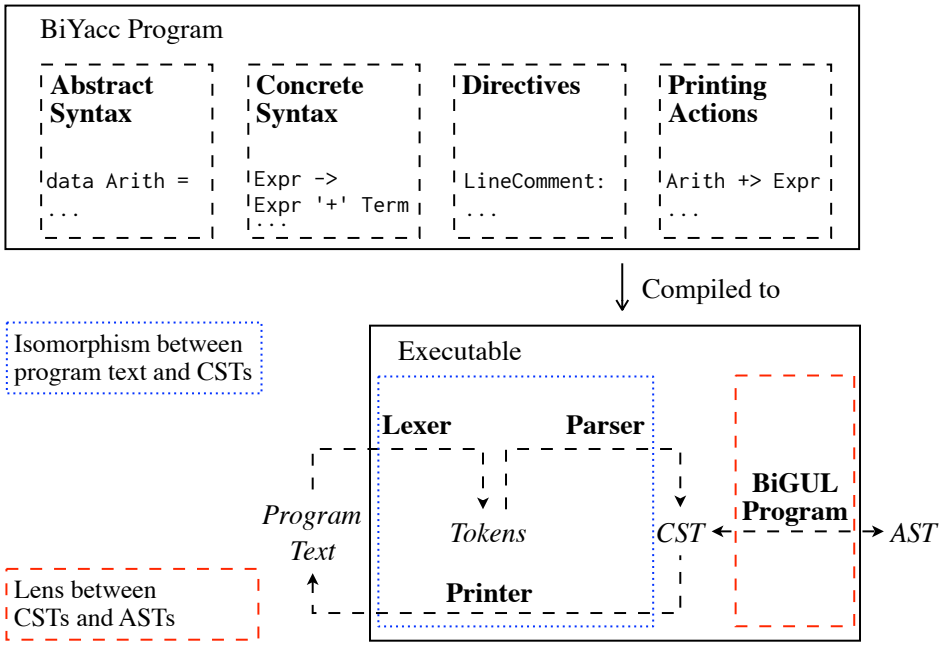


Figure 3.2: Architecture of BiYacc.

value is eventually replaced by the integer 2 of the AST, and a default layout (a space) is inserted after it. This is why the updated result for this part is '`2 `' instead of '`2`'. For the update on `(a)` with `Var "a"`, the actions succeed at lines 35, 41, 30, 35, and 40.

3.2 Design Details

In this section, we expound on a basic version of BiYacc that handles unambiguous grammars. The architecture is illustrated in Figure 3.2, where a BiYacc program

$$\text{`\texttt{\#Abstract}'}\ \mathit{decls}\ \text{`\texttt{\#Concrete}'}\ \mathit{pgs}\ \text{`\texttt{\#Directives}'}\ \mathit{drtvs}\ \text{`\texttt{\#Actions}'}\ \mathit{ags}\,, \tag{3.1}$$

consisting of abstract syntax, concrete syntax, directives, and printing actions, as formally defined in Figure 3.3, is compiled into into a few Haskell source files and then into an executable (by a Haskell compiler) for converting between program text and ASTs. Specifically:

- The abstract syntax part (*decls* for Haskell data type declarations) is already valid Haskell code and is (almost) directly used as the definitions of AST data types.

- The concrete syntax part (*pgs* for production groups) is translated to definitions of CST data types (whose elements are representations of how a string is produced using the production rules), and also used to generate the pair of *concrete* parser (including a lexer) and printer for the conversion between program text and CSTs. This pair of concrete parser and printer can be shown to form a (partial) *isomorphism* (which will be defined in Section 2.2.1). This part will be explained in Section 3.2.2.
- The directives part (*drctvs* for directives) is used in the lexer for recognising single-line and multi-line comments.
- The printing actions part (*ags* for action groups) is translated to a BiGUL program (which is a lens, see Definition 2.2.2) for handling (the semantic part of) parsing and retentive printing between CSTs and ASTs. This part will be explained in Section 3.2.3.

The whole executable is a well-behaved lens since it is the composition of an isomorphism and a lens, as stated in Lemma 2.2.5.

3.2.1 The Underlying DSL, BiGUL

From a BiYacc program, in addition to generating a parser and a printer, we also need to guarantee that the two generated programs are consistent with each other, i.e. satisfying the partial version of the well-behavedness (Definition 2.2.2). This is achieved by compiling BiYacc programs to the put-based bidirectional programming language BiGUL[1] (Ko et al., 2016). It has been formally verified in Agda (Norell, 2007) that BiGUL programs always denote well-behaved lenses, and BiGUL has been ported to Haskell as an embedded DSL library (Hu and Ko, 2018). More details about BiGUL can be found in the lecture notes on BiGUL programming (Hu and Ko, 2018).

In this subsection, we briefly introduce three operations, `Replace`, `Update`, and `Case`, which are used in BiGUL programs compiled from BiYacc programs. A BiGUL program has type BiGUL s v, where s and v are respectively the source and view types.

Replace. The simplest BiGUL operation we use is

$$\text{Replace} :: \text{BiGUL}\ s\ s$$

which discards the original source and returns the view—which has the same type as the source—as the updated source. That is, the *put* semantics of Replace is the function $\lambda s\ v \rightarrow \text{Just}\ v$.

[1] The work on BiGUL is not a contribution of this thesis.

Program ::= '`#Abstract`' *HsDeclarations*
['`#Concrete`' *ProductionGroup*$^+$]
'`#Directives`' *CommentSyntaxDecl Disambiguation*
'`#Actions`' *ActionGroup*$^+$
['`#OtherFilters`' *OtherFilters*]

ProductionGroup ::= *Nonterminal* '`->`' *ProductionBody*$^+${'`|`'} '`;`'

ProductionBody ::= ['`[`'*Constructor*'`]`'] *Symbol*$^+$ ['`{#`'`Bracket`'`#}`']

Symbol ::= *Primitive* | *Terminal* | *Nonterminal*

Constructor ::= *Nonterminal*

CommentSyntaxDecl ::= '`LineComment:`' *String* '`;`' '`BlockComment:`' *String* '`;`'

Disambiguation ::= [*Priority*] [*Associativity*]

ActionGroup ::= *HsType* '`+>`' *Nonterminal*
Action$^+$ '`;;`'

Action ::= *HsPattern* '`+>`' *Update*$^+$ '`;`'

Update ::= *Symbol*
| '`[`' *HsVariable* '`+>`' *UpdateCondition* '`]`'
| '`(`' *Nonterminal* '`->`' *Update*$^+$ '`)`'

UpdateCondition ::= *Symbol*
| '`(`' *Nonterminal* '`->`' *UpdateCondition*$^+$ '`)`'

Figure 3.3: Syntax of BiYacc programs. (Nonterminals with prefix *Hs* denote Haskell entities and follow the Haskell syntax; the notation $nt^+\{sep\}$ denotes a nonempty sequence of the same nonterminal *nt* separated by *sep*. Optional elements are enclosed in a pair of square brackets. The parts relating to *disambiguation* and *filters* will be explained in Chapter 4.)

Update. The next operation `update` is more complex, and is implemented with the help of Template Haskell (Sheard and Jones, 2002). The general form of the operation is

$$\$(update\ [p|\ spat\ |]\ [p|\ vpat\ |]\ [d|\ bs\ |]) :: \text{BiGUL}\ s\ v\ .$$

This operation decomposes the source and view by pattern matching with the patterns *spat* and *vpat* respectively, pairs the source and view components as specified by the patterns (see below), and performs further BiGUL operations listed in *bs* on the source–view pairs; the way to determine which source and view components are paired and which operation is performed on a pair is by looking for the same names in the three arguments. For example, the `update` operation

```
$(update [p| (x, _) |] [p| x |] [d| x = Replace |])
```

matches the source with a tuple pattern `(x, _)` and the view with a variable pattern `x`, so that the first component of the source tuple is related with the whole view; during the update, the first component of the source is replaced by the whole view, as indicated by the operation `x = Replace`. (The part marked by underscore (_) simply means that it will be skipped during the update.) Given a source `(1,2)` and a view `3`, the operation will produce `(3,2)` as the updated source. In general, any (type-correct) BiGUL program can be used in the list of further updates, not just the primitive Replace.

Case. The most complex operation we use is Case for doing case analysis on the source and view:

$$\mathsf{Case} :: [\mathsf{Branch}\ s\ v] \to \mathsf{BiGUL}\ s\ v\ .$$

Case takes a list of branches, of which there are two kinds: *normal* branches and *adaptive* branches. For a normal branch, we should specify a main condition using a source pattern *spat* and a view pattern *vpat*, and an exit condition using a source pattern *spat′*:

$$\$(\mathit{normalSV}\ [p|\ \mathit{spat}\ |]\ [p|\ \mathit{vpat}\ |]\ [p|\ \mathit{spat}'\ |]) :: \mathsf{BiGUL}\ s\ v \to \mathsf{Branch}\ s\ v\ .$$

An adaptive branch, on the other hand, only needs a main condition:

$$\$(\mathit{adaptiveSV}\ [p|\ \mathit{spat}\ |]\ [p|\ \mathit{vpat}\ |]) :: (s \to v \to s) \to \mathsf{BiGUL}\ s\ v\ .$$

Their semantics in the *put* direction are as follows: A branch is applicable when the source and view respectively match *spat* and *vpat* in its main condition. Execution of a Case chooses the first applicable branch from the list of branches, and continues with that branch. When the applicable branch is a normal branch, the associated BiGUL operation is performed, and the updated source should satisfy the exit condition *spat′* (or otherwise execution fails); when the applicable branch is an adaptive branch, the associated function is applied to the source and view to compute an adapted source, and the whole Case is rerun on the adapted source and the view; it must go into a normal branch this time, otherwise the execution fails. Think of an adaptive branch as bringing a source that is too mismatched with the view to a suitable shape—for example, when the source is a subtraction while the view is an addition, which are by no means in correspondence, we must adapt the source to an addition—so that a normal branch that deals with sources and views in some sort of correspondence can take over. This adaptation mechanism is used by BiYacc to print an AST when the source program text is too different from the AST or even nonexistent at all.

3.2.2 The Concrete Parsing and Printing Isomorphism

In this subsection, we describe the generation of CST data types and concrete printers (Section 3.2.2.1), the generation of concrete parsers (Section 3.2.2.2), and finally the inverse properties satisfied by the concrete parsers and printers (Section 3.2.2.3).

3.2.2.1 Generating CST Data Types and Concrete Printers

In BiYacc, we represent CSTs produced from a nonterminal *nt* as an automatically generated Haskell datatype named *nt*, whose constructors represent the production rules for *nt*. For each of these data types, we also generate a printing function which takes a CST as input and produces a string as dictated by the production rules in the CST.

For instance, in Figure 3.1, the group of production rules from the nonterminal `Factor` (lines 18–21) is translated to the following Haskell datatype and concrete printing function:

```
data Factor = Factor1 String Factor
            | Factor2 (String, String)
            | Factor3 (String, String)
            | Factor4 String Expr String
            | FactorNull

cprtFactor :: Factor -> String
cprtFactor (Factor1 s1 factor1)       = "-" ++ s1 ++ cprtFactor factor1
cprtFactor (Factor2 (numeric, s1))    = numeric ++ s1
cprtFactor (Factor3 (identifier, s1)) = identifier ++ s1
cprtFactor (Factor4 s1 expr1 s2)      = "(" ++ s1 ++ cprtExpr expr1 ++ ")" ++ s2
cprtFactor FactorNull                 = ""

cprtExpr :: Expr -> String
...
```

where `Factor1` ...`Factor4` are constructors corresponding to the four production rules, while `FactorNull` represents an empty CST of type *Factor* and is used as the default value whenever we want to create new program text depending on the view only. As an example, `Factor1` represents the production rule `Factor -> '-' Factor`, and its `String` field stores the whitespace appearing after a negation sign in the program text.

Following this idea, we define the translation from production rule groups (*pgs* in formula (3.1))

to datatype definitions by source-to-source compilation rules:

$$
\begin{aligned}
&[\![pgs]\!]_{ProductionGroup} = \langle [\![pg]\!]_{ProductionGroup} \mid pg \in pgs \rangle \\
&[\![nt \text{ `->'} bodies]\!]_{ProductionGroup} = \\
&\quad \texttt{`data'}\ nt\ \texttt{`='} \langle \textsc{con}(nt, body)\ \langle \textsc{field}(s) \mid s \in body \rangle\ \texttt{`|'} \mid body \in bodies \rangle \\
&\qquad \textsc{nullCon}(nt)\ .
\end{aligned}
$$

Compilation rules of this kind will also be used later, so we introduce the notation here: Compilation rules are denoted by semantic brackets ($[\![\cdot]\!]$), and refer to some auxiliary functions, whose names are in SMALL CAPS. A nonterminal in subscript gives the 'type' of the argument or metavariable before it. The angle bracket notation $\langle f\ e \mid e \in es \rangle$ denotes the generation of a list of entities of the form $f\ e$ for each element e in the list es, in the order of their appearance in es. In more detail: $\textsc{con}(nt, body)$ retrieves the constructor for a production rule. The fields of a constructor are generated from the right-hand side of the corresponding production rule in the way described by the auxiliary function FIELD— nonterminals that are not primitives are left unchanged (using their names for data types), primitives are stored in the `String` type [1], terminal symbols are dropped, and an additional `String` field is added for each terminal and primitive for storing layout information (whitespaces and comments) appearing after the terminal or primitive in the program text. The last step is to insert an additional empty constructor, whose name is denoted by $\textsc{nullCon}(nt)$.

3.2.2.2 Generating Concrete Lexers and Parsers

The implementation of the concrete parser, which turns program text into CSTs, is further separated into two phases: lexing and parsing. In both phases, the layout information (whitespaces and comments) is automatically preserved, which makes the CSTs isomorphic to the program text.

Lexer. Apart from handling the terminal symbols appearing in a grammar, the lexer automatically derived by BiYacc can also recognise several kinds of literals, including integers, strings, and identifiers, respectively produced by the nonterminals `Numeric`, `String`, and `Identifier`. For now, the forms of these literals are predefined, but we take this as a step towards a lexerless grammar, in which strings produced by nonterminals can be specified in terms of regular expressions. Furthermore, whitespaces and comments are carefully handled in the derived lexer, so they can be

[1] The reason for storing primitives in the String type is because String is the most precise representation that will not cause the loss of any information. For instance, this is useful for retaining the leading zeros of an integer such as 073. Storing 073 as Integer will cause the loss of the leading zero.

completely stored in CSTs and correctly recovered to the program text in printing. This feature of BiYacc, which we explain below, makes layout preservation transparent to the programmer.

An assumption of BiYacc is that whitespaces are only considered as separators between other tokens. (Although there exist some languages such as Haskell and Python where indentation does affect the meaning of a program, there are workarounds, e.g. writing a preprocessing program to insert explicit separators.) Usually, token separators are thrown away in the lexing phase, but since we want to keep layout information in CSTs, which are built by the parser, the lexer should leave the separators intact and pass them to the parser. The specific approach taken by BiYacc is wrapping a lexeme and the whitespaces following it into a single token. Beginning whitespaces are treated separately from lexing and parsing, and are always preserved. And in this prototype implementation, comments are also considered as whitespaces.

Parser. The concrete parser is used to generate a CST from a list of tokens according to the production rules in the grammar. Our parser is built using the parser generator Happy (Marlow and Gill, 2001), which takes a specification of a grammar in Backus normal form (BNF) with semantic actions and produces a Haskell module containing a parser function. The grammar we feed into Happy is still essentially the one specified in a BiYacc program, but in addition to parsing and constructing CSTs, the Happy actions also transfer the whitespaces wrapped in tokens to corresponding places in the CSTs. For example, the production rules for `Factor` in the expression example, as shown on the left below, are translated to the Happy specification on the right:

```
Factor                          Factor
   -> '-' Factor                  : token1 Factor       { Factor1 $1 $2 }
   |  Numeric                     | tokenNumeric        { Factor2 $1 }
   |  Identifier          ⇝       | tokenIdentifier     { Factor3 $1 }
   |  '(' Expr ')' ;              | token2 Expr token3  { Factor4 $1 $2 $3 } .
```

We use the first expansion (`token1 Factor`) to explain how whitespaces are transferred: The generated Happy token `token1` matches a '`-`' token produced by the lexer, and extracts the whitespaces wrapped in the '`-`' token; these whitespaces are bound to `$1`, which is placed into the first field of `Factor1` by the associated Haskell action.

3.2.2.3 Inverse Properties

Now we give the types of the concrete printer and parser generated from a BiYacc program and show that they form an isomorphism. Let the type CST be the set of all the CSTs defined by the grammar of a BiYacc program; by default it is the source type (nonterminal) of the first group

of actions in the `#Actions` part. We have seen in Section 3.2.2.1 how to generate its datatype definition and a concrete printing function

$$\mathit{cprint} :: \mathsf{CST} \to \mathsf{String} .$$

On the other hand, from the grammar we directly use a parser generator to generate a concrete parsing function

$$\mathit{cparse} :: \mathsf{String} \to \mathsf{Maybe\ CST} ,$$

which is Maybe-valued since a piece of input text may be invalid. This *cparse* function is one direction of the isomorphism in the executable, while the other direction is

$$\mathsf{Just} \circ \mathit{cprint} :: \mathsf{CST} \to \mathsf{Maybe\ String} .$$

Below we show that the inverse properties amount to the requirements that the generated parser is 'correct' and the grammar is unambiguous.

Since our concrete parsers are generated by the parser generator Happy (Marlow and Gill, 2001), we need to assume that they satisfy some essential properties, for we cannot control the generation process and verify those properties.

Definition 3.2.1 (Parser Correctness)**.** A parser *cparse* is *correct* with respect to a printer *cprint* exactly when

$$\mathit{cparse}\ \mathit{text} = \mathsf{Just}\ \mathit{cst} \quad \Rightarrow \quad \mathit{cprint}\ \mathit{cst} = \mathit{text} \tag{3.2}$$

$$\mathit{cprint}\ \mathit{cst} = \mathit{text} \quad \Rightarrow \quad \exists\ \mathit{cst}'.\ \mathit{cparse}\ \mathit{text} = \mathsf{Just}\ \mathit{cst}' . \tag{3.3}$$

To see what (3.2) means, recall that our CSTs, as described in Section 3.2.2.1, encode precisely the derivation trees, with the CST constructors representing the production rules used, and *cprint* traverses the CSTs and follows the encoded production rules to produce the derived program text. Now consider what *cparse* is supposed to do: It should take a piece of program text and find a derivation tree for it, i.e. a CST which *cprint*s to that piece of program text. This statement is exactly (3.2). In other words, (3.2) is the functional specification of parsing, which is satisfied if the parser generator we use behaves correctly. Also it is reasonable to expect that a parser will be able to successfully parse any valid program text, and this is exactly (3.3).

We also need to make an assumption about concrete printers: throughout this section we assume that the grammar is unambiguous, and this amounts to injectivity of *cprint*—for any piece

of program text there is at most one CST that prints to it.

With these assumptions, we can now establish the isomorphism (which is rather straightforward).

Theorem 3.2.2 (Inverse Properties)**.** If a parser *cparse* is correct with respect to an injective printer *cprint*, then *cparse* and Just ∘ *cprint* form an isomorphism, that is,

$$cparse\ text = \mathsf{Just}\ cst \quad \Leftrightarrow \quad (\mathsf{Just} \circ cprint)\ cst = \mathsf{Just}\ text\ .$$

Proof. The left-to-right direction is immediate since the right-hand side is equivalent to *cprint cst* = *text*, and the whole implication is precisely (3.2). For the right-to-left direction, again the antecedent is equivalent to *cprint cst* = *text*, and we can invoke (3.3) to obtain *cparse* (*text*) = Just *cst′* for some *cst′*. This is already close to our goal—what remains to be shown is that *cst′* is exactly *cst*, which is indeed the case because

$$\begin{aligned}
&cparse\ text = \mathsf{Just}\ cst' \\
\Rightarrow\quad &\{\ \text{antecedent}\ \} \\
&cparse\ (cprint\ cst) = \mathsf{Just}\ cst' \\
\Rightarrow\quad &\{\ (3.2)\ \} \\
&cprint\ cst' = cprint\ cst \\
\Rightarrow\quad &\{\ cprint\ \text{is injective}\ \} \\
&cst' = cst\ .
\end{aligned}$$

□

3.2.3 Generating the BiGUL Lens

The source-to-source compilation from the actions part of a BiYacc program to a BiGUL program (i.e. lens) is shown in Figure 3.4 and Figure 3.5. Additional arguments to the semantic bracket are typeset in superscript, and the notation $\langle \ldots \mid \ldots \in \ldots \rangle\{s\}$ means inserting s between the elements of the list.

Action Groups. Each group of actions is translated into a small BiGUL program, whose name is determined by the view type *vt* and source type *st* and denoted by $\textsc{prog}(vt, st)$. The BiGUL program has one single `Case` statement, and each action is translated into two branches in this `Case` statement, one normal and the other adaptive. All the adaptive branches are gathered in the

$$
\begin{aligned}
&[\![\text{`\#Abstract'}\ decls\ \text{`\#Concrete'}\ pgs\ \text{`\#Directives'}\ drctvs\ \text{`\#Actions'}\ ags_{Program}]\!] = \\
&\quad decls\ \langle [\![pg]\!]_{ProductionGroup} \mid pg \in pgs \rangle\ \langle [\![ag]\!]_{ActionGroup} \mid ag \in ags \rangle \\
&[\![vt\ \text{`+>'}\ st\ acts]\!]_{ActionGroup} = \textsc{prog}(vt, st)\ \text{`::'}\ \texttt{BiGUL}\ st\ vt \\
&\quad \textsc{prog}(vt, st)\ \text{`='}\ \texttt{Case} \\
&\quad\quad \text{`[']}\ \langle [\![a]\!]^{\mathsf{N}, vt, st}_{Action}\ \text{`,'} \mid a \in acts \rangle\ \langle [\![a]\!]^{\mathsf{A}, st}_{Action} \mid a \in acts \rangle \{\text{`,'}\}\ \text{`]'} \\
&[\![vpat\ \text{`+>'}\ updates]\!]^{\mathsf{N}, vt, st}_{Action} = \\
&\text{`\$(normalSV'} \\
&\quad \text{`[p|']}\ \textsc{srcCond}(\textsc{ersVars}(\text{`['}\ st\ \text{`->'}\ updates\ \text{`]'})_{Update})\ \text{`|]'}\ \text{`[p|']}\ vpat\ \text{`|]'} \\
&\quad \text{`[p|']}\ \textsc{srcCond}(\textsc{ersVars}(\text{`('}\ st\ \text{`->'}\ updates\ \text{`)'})_{Update})\ \text{`|])'} \\
&\quad\quad \text{`\$(update'}\ \text{`[p|']}\ \textsc{removeAs}(vpat)\ \text{`|]'} \\
&\quad\quad\quad\quad \text{`[p|']}\ \textsc{srcPat}(\text{`('}\ st\ \text{`->'}\ updates\ \text{`)'})_{Update}\ \text{`|]'} \\
&\quad\quad\quad\quad \text{`[d|']}\ \langle [\![u]\!]^{vt, vpat}_{Update} \mid u \in updates \rangle\ \text{`|])'} \\
&[\![\text{`['}\ var\ \text{`+>'}\ uc_{Primitive}\ \text{`]'}]\!]^{vt, vpat}_{Update} = var\ \text{`= Replace;'} \\
&[\![\text{`['}\ var\ \text{`+>'}\ uc_{Nonterminal}\ \text{`]'}]\!]^{vt, vpat}_{Update} = var\ \text{`='}\ \textsc{prog}(\textsc{varType}(vt, vpat, var), uc)\ \text{`;'} \\
&[\![\text{`['}\ var\ \text{`+>'}\ \text{`('}\ nt\ \text{`->'}\ \ldots\ \text{`)'}\ \text{`]'}]\!]^{vt, vpat}_{Update} = [\![\text{`['}\ var\ \text{`+>'}\ nt\ \text{`]'}]\!]^{vt, vpat}_{Update} \\
&[\![\text{`('}\ \ldots\ \text{`->'}\ updates\ \text{`)'}]\!]^{vt, vpat}_{Update} = \langle [\![u]\!]^{vt, vpat}_{Update}\ \text{`;'} \mid u \in updates \rangle \\
&[\![symbol]\!]^{vt, vpat}_{Update} = \text{``} \\
&[\![vpat\ \text{`+>'}\ updates]\!]^{\mathsf{A}, st}_{Action} = \text{`\$(adaptiveSV'}\ \text{`[p| \_ |]'}\ \text{`[p|']}\ vpat\ \text{`|])'} \\
&\quad\quad\quad\quad\quad\quad\quad\quad \text{`(\textbackslash\_ \_ ->'}\ \textsc{defaultExpr}(\textsc{ersVars}(\text{`('}\ st\ \text{`->'}\ updates\ \text{`)'}))
\end{aligned}
$$

Figure 3.4: Semantics of BiYacc programs (as BiGUL programs).

second half of the `Case` statement, so that normal branches will be tried first. For example, the third group of type `Arith +> Factor` is compiled to

```
bigulArithFactor :: BiGUL Factor Arith
bigulArithFactor =
  Case [ ... -- normal branches
         ... -- adaptive branches
       ] .
```

Normal Branches. We said in Section 3.1 that the semantics of an action is to perform pattern matching on both the source and view, and then update parts of the source with parts of the view. This semantics is implemented with a normal branch: The source and view patterns are compiled to the main condition, and, together with the updates overlaid on the source pattern, also to an

$$
\begin{aligned}
&\textsc{field}(nt)_{Nonterminal} = nt \\
&\textsc{field}(t)_{Terminal} = \text{`String'} \\
&\textsc{field}(p)_{Primitive} = \text{`(' } p \text{ `, String)'} \\
&\textsc{ersVars}(\text{`[' } var \text{ `+>' } uc \text{ `]'})_{Update} = uc \\
&\textsc{ersVars}(\text{`(' } nt \text{ `->' } updates \text{ `)'})_{Update} = \text{`(' } nt \text{ `->' } \langle \textsc{ersVars}(u) \mid u \in updates \rangle \text{ `)'} \\
&\textsc{ersVars}(symbol)_{Update} = symbol \\
&\textsc{srcCond}(\text{`(' } nt \text{ `->' } uconds \text{ `)'})_{UpdateCondition} = \text{`(' } \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds \rangle) \\
&\qquad \langle \textsc{srcCond}(uc) \mid uc \in uconds \rangle \text{ `)'} \\
&\textsc{srcCond}(symbol)_{UpdateCondition} = \text{`\_'} \\
&\textsc{condHead}(\text{`(' } nt \text{ `->' } \ldots \text{ `)'})_{UpdateCondition} = nt \\
&\textsc{condHead}(symbol)_{UpdateCondition} = symbol \\
&\textsc{srcPat}(\text{`[' } var \text{ `+>' } uc_{Primitive} \text{ `]'})_{Update} = \text{`(' } var \text{ `, \_)'} \\
&\textsc{srcPat}(\text{`[' } var \text{ `+>' } uc_{Nonterminal} \text{ `]'})_{Update} = var \\
&\textsc{srcPat}(\text{`(' } nt \text{ `->' } updates \text{ `)'})_{Update} = \text{`(' } \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in \textsc{ersVars}(updates) \rangle) \\
&\qquad \langle \textsc{srcPat}(u) \mid u \in updates \rangle \text{ `)'} \\
&\textsc{srcPat}(symbol)_{Symbol} = \text{`\_'} \\
&\textsc{defaultExpr}(symbol)_{Primitive} = \text{`(undefined, " ")'} \\
&\textsc{defaultExpr}(symbol)_{Nonterminal} = \textsc{nullCon}(symbol) \\
&\textsc{defaultExpr}(symbol)_{Terminal} = \text{`" "'} \\
&\textsc{defaultExpr}(\text{`(' } nt \text{ `->' } uconds \text{ `)'})_{UpdateCondition} = \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds \rangle) \\
&\qquad \langle \textsc{defaultExpr}(uc) \mid uc \in uconds \rangle
\end{aligned}
$$

Figure 3.5: Semantics of BiYacc programs (as BiGUL programs)—auxiliary functions.

`update` operation. For example, the first action in the `Arith-Factor` group

```
Sub (Num 0) y +> '-' (y +> Factor)
```

is compiled to

```
$(normalSV [p| (Factor1 _ _) |] [p| Sub (Num 0) y |] [p| (Factor1 _ _) |])
  $(update [p| Sub (Num 0) y |] [p| (Factor1 _ y) |] [d| y = bigulArithFactor; |]) .
```

When the CST is a `Factor1` and the AST matches `Sub (Num 0) y`, we enter this branch, decompose the source and view by pattern matching, and use the view's right subtree `y` to update the second field of the source while skipping the first field (which stores whitespaces); the name of the BiGUL program for performing the update is determined by the type of the smaller source `y` (deduced by varType) and that of the smaller view.

Adaptive Branches. When all actions in a group fail to match, we should adapt the source into a proper shape to correspond to the view. This is done by generating adaptive branches from the actions during compilation. For example, besides a normal branch, the first action in the `Arith–Factor` group `Sub (Num 0) y +> '-' (y +> Factor)` is also compiled to

```
$(adaptiveSV [p| _ |] [p| Sub (Num 0) _ |]) (\ _ _ ->  Factor1 " " FactorNull) .
```

Since the source pattern of the main condition (of the adaptive branch) is a wildcard, the branch is always applicable if the view matches `Sub (Num 0) _`. The body of the adaptation function is generated by the auxiliary function DEFAULTEXPR, which creates a skeletal value—here `Factor1 " " FactorNull` represents a negation skeleton `- ` whose value is not (recursively) created yet—that matches the source pattern. These adaptive branches are placed at the end of an action group and tried only if no normal branches are applicable so that unnecessary adaptation will never be performed.

Entry Point. The entry point of the program is chosen to be the BiGUL program compiled from the first group of actions. This corresponds to our assumption that the initial input concrete and abstract syntax trees are of the types specified for the first action group. It is rather simple so the rules are not shown in the figure. For the expression example, we generate a definition

```
entrance = bigulArithExpr
```

which is invoked in the `main` program.

Well-behavedness. Since BiGUL programs always denote well-behaved lenses, a fact which has been formally verified (Norell, 2007), we get the following theorem for free.

Theorem 3.2.3 (Well-behavedness)**.** The BiGUL program generated from a BiYacc program is a lens; that is, it satisfies the well-behavedness laws in Definition 2.2.2 with *cst* substituted for the source *s* and *ast* for the view *v*:

$$\begin{aligned} \mathit{put}\ \mathit{cst}\ \mathit{ast} = \mathsf{Just}\ \mathit{cst}' \quad &\Rightarrow \quad \mathit{get}\ \mathit{cst}' = \mathsf{Just}\ \mathit{ast} \\ \mathit{get}\ \mathit{cst} = \mathsf{Just}\ \mathit{ast} \quad &\Rightarrow \quad \mathit{put}\ \mathit{cst}\ \mathit{ast} = \mathsf{Just}\ \mathit{cst}\ . \end{aligned}$$

3.3 Case Study: The Tiger Language

The design of BiYacc may look simplistic and make the reader wonder how much it can describe. In fact, BiYacc can already handle real-world language features: We successfully built a pair of

parser and printer for almost a full set of the C programming language following the C89 grammar based on Kernighan and Ritchie (1989) (excluding preprocessing parts and several primitive types). As another example, Kinoshita and Nakano (2017) adopted BiYacc as part of their system for synchronising Coq functions and their corresponding Ocaml programs.

But due to the fact that we could not find any official definition of the abstract syntax of C, in this section, we will demonstrate BiYacc with a medium-size case study on the Tiger language, which is a statically typed imperative language first introduced in Appel's textbook on compiler construction (Appel, 1998). Since Tiger's purpose of design is pedagogical, it is not too complex and yet covers many important language features including conditionals, loops, variable declarations and assignments, and function definitions and calls. Tiger is therefore a good case study with which we can test the potential of our BX-based approach to constructing parsers and retentive printers. Some of these features can be seen in this Tiger program:

```
function foo() =
  (for i := 0 to 10
    do (print(if i < 5 then "smaller"
                       else "bigger");
        print("\n"))) .
```

To give a sense of Tiger's complexity, it takes a grammar with 81 production rules to specify Tiger's syntax, while for C89 and C99 it takes respectively 183 and 237 rules[1] without any disambiguation declarations (based on Kernighan and Ritchie (1989) and the draft version of 1999 ISO C standard, excluding the preprocessing part). The difference is basically due to the fact that C has more primitive types and various kinds of assignment statements.

Excerpts of the abstract and concrete syntax of Tiger are shown in Figure 3.6. The abstract syntax is largely the same as the original one defined in Appel's textbook (page 98); as for the concrete syntax, Appel does not specify the whole grammar in detail, so we use a version slightly adapted from Hirzel and Rose's lecture notes (Hirzel and Rose, 2013). In order to make the grammar unambiguous, we divide the (binary) operators into several groups, with the highest-precedence terms (like literals) placed in the last group, just like what we did in the arithmetic expression example (Figure 3.1); to handle features that are not supported by BiYacc, we let the AST constructors `TFunctionDec` and `TTypeDec` take a single function and type declaration respectively,

[1] However, the grammar uses extended BNF and many production rules require an expansion regarding their optional cases in order to be accepted by Happy. For instance, we need to fully expand *for* ($Expr_{opt}$; $Expr_{opt}$; $Expr_{opt}$) and write a total of eight cases regarding the occurrence or absence of each expression. In this sense, C has a lot more production rules than Tiger has.

instead of a list of adjacent ones (for representing mutual recursion) as in Appel (1998), since we cannot handle the synchronisation between a list of lists (in ASTs) and a list (in CSTs) with BiYacc's syntax; finally, to circumvent the 'dangling else problem', a terminal 'end' is added to mark the end of an `if-then` expression.

We have successfully tested our BiYacc program for Tiger on all the sample programs provided on the homepage of Appel's book[1], including a merge sort implementation and an eight-queen solver, and there is no problem parsing and printing them with well-behavedness guaranteed. In the following subsections, we will present some printing strategies described in the BiYacc program to demonstrate what BiYacc, in particular retentive printing, can achieve.

3.3.1 Syntactic Sugar and Resugaring

We start with a simple example about syntactic sugar, which is pervasive in programming languages and lets the programmer use some features in an alternative (perhaps conceptually higher-level) syntax. For instance, Tiger represents boolean values false and true respectively as zero and nonzero integers, and the logical operators `&` ('and') and `|` ('or') are converted to a conditional structure in the abstract syntax: `e1 & e2` is desugared and parsed to `TCond e1 e2 (TInt 0)` and `e1 | e2` to `TCond e1 (TInt 1) e2`. The printing actions for them in BiYacc are

```
TExp +> Prmtv
  TCond e1 (TInt 1) (JJ e2)  +>  [e1 +> Prmtv] '|' [e2 +> Prmtv1];

TExp +> Prmtv1
  TCond e1 e2 (JJ (TInt 0))  +>  [e1 +> Prmtv1] '&' [e2 +> Prmtv2]; .
```

The *parse* function for these kinds of syntactic sugar is not injective, since the basic syntax and its sugared form are both mapped to the same AST structure. A conventional printer which takes only the AST as input cannot reliably determine whether an abstract expression should be printed to the basic form or the sugared form, whereas a retentive printer can make the correct decision by inspecting the CST.

The idea of *resugaring* (Pombrio and Krishnamurthi, 2014) is to print evaluation sequences in a core language in terms of a surface syntax. Here we show that, without any extension, BiYacc is already capable of propagating some AST changes that result from evaluation back to the concrete syntax, subsuming a part of Pombrio and Krishnamurthi's work (Pombrio and Krishnamurthi, 2014, 2015).

[1] https://www.cs.princeton.edu/~appel/modern/testcases/

```
#Abstract
type TSymbol = String                      data BBool = TT | FF
                                           data MMaybe a = NN | JJ a
data Tuple a b = Tuple a b                 data List a = Nil | Cons a (List a)

data TExp = TString String | TInt Int  | TNilExp | TCond TExp TExp (MMaybe TExp)
  | TLet (List TDec) TExp | TOp TExp TOper TExp | TExpSeq (List TExp) | ...

data TOper = TPlusOp | TMinusOp | ... | TEqOp | TNeqOp | ...

data TDec = TVarDec TSymbol BBool (MMaybe TSymbol) TExp
          | TTypeDec (Tuple TSymbol TTy) | TFunctionDec TFundec

data TFundec = TFundec TSymbol (List TFieldDec) (MMaybe TSymbol) TExp ...

#Concrete
Exp ->  LetExp | ArrExp | IfThen   | IfThenElse | Prmtv
     |  ForExp | RecExp | WhileExp | Assignment | 'break' ;

VarDec -> 'var' Identifier               ':=' Exp
        | 'var' Identifier ':' Identifier ':=' Exp ;

LValue      -> Identifier  |  OtherLValue ;
OtherLValue -> LValue '.' Identifier
  |  Identifier '[' Exp ']' | OtherLValue '[' Exp ']'  ;

SeqExp  -> '(' ')'        | '{' ExpSeq '}' ;
ExpSeq  -> Exp ';' ExpSeq | Exp ;

IfThenElse -> [ITE] 'if' Exp 'then' Exp 'else' Exp ;
IfThen     -> [IT]  'if' Exp 'then' Exp 'end' ;

Prmtv  -> Prmtv '|'  Prmtv1        Prmtv1 -> Prmtv1 '&' Prmtv2
        | Prmtv1 ;                         | Prmtv2 ;

Prmtv3 -> Prmtv3 '+'  Prmtv4       Prmtv5 -> '-' Prmtv5 | Numeric | String
        | Prmtv3 '-'  Prmtv4               | LValue | SeqExp | CallExp | "nil" ;
        | Prmtv4 ; ...             ...
```

Figure 3.6: An excerpt of Tiger's abstract and unambiguous concrete syntax. (Here we define our own BBool type and MMaybe type for avoiding name clashing with Haskell's built-in ones.)

We borrow their example of resugaring evaluation sequences for the logical operators 'or' and 'not', but recast the example in Tiger. The 'or' operator has been defined as syntactic sugar in Section 3.3.1. For the 'not' operator, which Tiger lacks, we introduce '~', represented by `TNot` in the abstract syntax. Now consider the source expression

```
~1 | ~0 ,
```

which is parsed to

```
TCond (TNot (TInt 1)) (TInt 1) (JJ (TNot (TInt 0))) .
```

A typical evaluator will produce the following evaluation sequence given the above AST:

```
  TCond (TNot (TInt 1)) (TInt 1) (JJ (TNot (TInt 0)))
→ TCond (TInt 0) (TInt 1) (JJ (TNot (TInt 0)))
→ TNot (TInt 0)
→ TInt 1 .
```

If we perform retentive printing after every evaluation step using BiYacc, we will get the following evaluation sequence on the source:

```
~1 | ~0   →   0 | ~0   →   ~0   →   1 .
```

Due to the Correctness property, parsing these concrete terms will yield the corresponding abstract terms in the abstract evaluation sequence, and this is exactly Pombrio and Krishnamurthi's 'emulation' property, which they have to prove for their system. For BiYacc, however, the emulation property holds by construction, since BiYacc programs are always well-behaved. Another difference is that we do not need to insert additional information (such as tags) into an AST for recording which surface syntax structure a node comes from. One advantage of our approach is that we keep the abstract syntax pure, so that other tools—the evaluator in particular—can process the abstract syntax without being modified, whereas in Pombrio and Krishnamurthi's approach, the evaluator has to be adapted to work on the enriched abstract syntax.

Note that the above resugaring for Tiger is achieved for free—the programmer does not need to write additional or special code. In general, BiYacc can easily and reliably propagate AST changes that involve only 'simplification', i.e. replacing part of an AST with a simpler tree. Thus it should not be surprising that BiYacc can also propagate simplification-like optimisations such as constant propagation and dead code elimination and some refactoring transformations such as variable renaming and adding or removing parameters. We can achieve all these by using one 'general-purpose' BiYacc program, which does not need to be tailored for each application.

3.3.2 Language Evolution

When a language evolves, some new features of the language (e.g. the `foreach` loop introduced in Java 5 (Gosling et al., 2006)) can be implemented by desugaring to some existing features (e.g. ordinary `for` loops), so that the compiler back end and abstract syntax definition do not need to be extended to handle the new features. As a consequence, all the engineering work about optimising transformations or refactoring (Fowler and Beck, 1999) that has been developed for the abstract syntax remains valid.

Consider a kind of 'generalised-`if`' expression allowing more than two cases, resembling the alternative construct in the guarded command language (Dijkstra, 1975). We extend TIGER's concrete syntax with the following production rules:

```
Exp    -> ... | Guard | ... ;
Guard  -> 'guard' CaseBs 'end';
CaseBs -> CaseB CaseBs | CaseB ;
CaseB  -> LValue '=' Numeric '->' Exp ; .
```

For simplicity, we restrict the predicate produced by `CaseB` to the form `LValue '=' Numeric`, but in general the `Numeric` part can be any expression computing an integer. The retentive printing actions for this new construct can still be written within BIYACC, but require much deeper pattern matching:

```
TExp +> Guard
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing  +>
    'guard' (CaseBs -> (CaseB -> [lv +> LValue] '='
                                [i +> Numeric] '->' [e1 +> Exp])
            ) 'end';
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TCond _ _ _))  +>
    'guard' (CaseBs -> (CaseB -> [lv +> LValue] '='
                                [i +> Numeric] '->' [e1 +> Exp])
                       [if2 +> CaseBs]
            ) 'end';
;;
```

```
TExp +> CaseBs
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing  +>
    (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp]);
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TCond _ _ _))  +>
    (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp])
    [if2 +> CaseBs];
;; .
```

Although a little complex, these printing actions are in fact fairly straightforward: The first group of type `Tiger +> Guard` handles the enclosing `guard`–`end` pairs, distinguishes between single- and multi-branch cases, and delegates the latter case to the second group, which prints a list of branches recursively.

This is all we have to do—the corresponding parser is automatically derived and guaranteed to be consistent. Now `guard` expressions are desugared to nested `if` expressions in parsing and preserved in printing, and we can also resugar evaluation sequences on the ASTs to program text. For instance, the following `guard` expression

```
guard  choice = 1  ->  4
       choice = 2  ->  8
       choice = 3  ->  16  end
```

is parsed to

```
TCond (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (JJ
  (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
```

where `TSimpleVar` is shortened to `TSV`, and `choice` is shortened to `c`. Suppose that the value of the variable `choice` is 2. The evaluation sequence on the AST will then be

```
   TCond (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
     (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
→ TCond (TInt 0) (TInt 4) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
     (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
→ TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))
→ TCond (TInt 1) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))
→ TInt 8 .
```

And the translated evaluation sequence on the concrete expression will be

```
   guard  choice = 1  ->  4
          choice = 2  ->  8
          choice = 3  ->  16  end
↛
→ guard  choice = 2  ->  8
         choice = 3 -> 16 end
↛
→ 8 .
```

Retentive printing fails for the first and third steps (the program text becomes an *if-then-else* expression if we do printing at these steps), but this behaviour in fact conforms to Pombrio and Krishnamurthi's 'abstraction' property, which demands that core evaluation steps that make sense only in the core language must not be propagated to the surface. In our example, the first and third steps in the `TCond`-sequence evaluate the condition to a constant, but conditions in guard expressions are restricted to a specific form and cannot be a constant; evaluation of guard expressions thus has to proceed in bigger steps, throwing away or going into a branch in each step, which corresponds to two steps for `TCond`.

The reader may have noticed that, after the `guard` expression is reduced to two branches, the layout of the second branch is disrupted; this is because the second branch is in fact printed from scratch. In current BiYacc, the printing from an AST to a CST is accomplished by recursively performing pattern matching on both tree structures. This approach naturally comes with the disadvantage that the matching is mainly decided by the position of the nodes in the AST and CST. Consequently, a minor structural change on the AST may totally disrupt the matching between the AST and the CST. Nonetheless, the problem can theoretically and completely be solved by a new property *Retentiveness* proposed in Chapter 5.

3.3.3 Other Potential Applications

We conclude this section by shortly discussing several other potential applications. In general, (current) BiYacc can easily and reliably propagate AST changes that have local effect such as replacing part of an AST with a simpler tree, without destroying the layouts and comments of unaffected code. Thus it would not be surprising that BiYacc can also propagate (i) simplification-like optimisations such as constant folding and constant propagation and (ii) some code refactoring transformations such as variable renaming. All these functionalities are achieved for free by one 'general-purpose' BiYacc program, which does not need to be tailored for each application.

Summary of the Case Study. We program in BiYacc and obtain a consistent pair of parser and retentive printer for the Tiger language. Well-behavedness of the pair is tested against all the sample programs provided on the homepage of Appel's book. We demonstrate that BiYacc is able to preserve syntactic sugar and achieve resugaring using clean and compact ASTs without *any other effort*, which are not the case for conventional approaches (that choose to enrich ASTs). We also demonstrate that, using BiYacc, if Tiger evolves and there is a new kind of 'generalised-if' expression that is desugared to the same `TCond` construct, all we need to do is add several cases for the production rules and printing actions; the newly generated parser and retentive printer are still consistent and correctly handle the new syntactic sugar.

3.4 Related Work

3.4.1 Building Parser and Printer Pairs at Once

In Section 2.3.1, we have seen that existing approaches are not designed to handle the synchronisation between program text and ASTs. Here, we further compare our approach with others' in depth, especially Rendel and Ostermann's (2010) and Matsuda and Wang's (2018b) approaches, which also guarantee some properties between a parser and printer pair.

Both Rendel and Ostermann and Matsuda and Wang (2018a; 2018b) adopt a combinator-based approach[1] whereas we use a generator-based approach, where small components describing both parsing and printing are glued together to yield more sophisticated behaviour, and can guarantee properties similar to Theorem 3.2.2 with *cst* replaced by *ast* in the equations. (Let us call the variant version Theorem 1′, since it will be used quite often later.) In Rendel and Ostermann's system (called 'invertible syntax descriptions', which we shorten to ISDs henceforth), both the parsing and printing semantics are predefined in the combinators and consistency is guaranteed by their partial isomorphisms, whereas in Matsuda and Wang's system (called FliPpr), the combinators describing pretty printing are translated by a semantic-preserving transformation to a core syntax, which is further processed by their grammar-based inversion system (Matsuda et al., 2010) to realise the parsing semantics. Brabrand et al. (2008) present a tool XSugar that handles bijections between the XML syntax (representation) and any other syntax (representation) for the same language, guaranteeing that the syntax transformation is reversible. However, the essential factor that distinguishes our system from others is that the printer produced from a BiYacc program is

[1] Although they use different implementation techniques, we will not dive into them in our related work. See Matsuda and Wang's (Matsuda and Wang, 2018a) related work for a comparison.

retentive and can deal with synchronisation.

Although the above-mentioned systems are tailored for unifying parsing and printing, there are design differences. An ISD is more like a parser, while FliPpr lets the user describe a printer: To handle operator priorities, for example, the user of ISDs will assign priorities to different operators, consume parentheses, and use combinators such as `chainl` to handle left recursion in parsing, while the user of FliPpr will produce necessary parentheses according to the operator priorities. For BiYacc in this chapter which deals with unambiguous grammars only, the user defines a concrete syntax that has a hierarchical structure (e.g., `Expr`, `Term`, and `Factor`) to express operator priority, and write printing strategies to produce (preserve) necessary parentheses. The user of XSugar will also likely need to use such a hierarchical structure.

It is interesting to note that the part producing parentheses in FliPpr essentially corresponds to the hierarchical structure of grammars. For example, to handle arithmetic expressions in FliPpr, we can write:

```
ppr' i (Minus x y) =
 parensIf (i >= 6) $ group $
   ppr 5 x <> nest 2
     (line' <> text "-" <> space' <> ppr 6 y); .
```

FliPpr will automatically expand the definition and derive a group of `ppr_i` functions indexed by the priority integer `i`, corresponding to the hierarchical grammar structure. In other words, there is no need to specify the concrete grammar, which is already implicitly embedded in the printer program. This makes FliPpr programs neat and concise. Following this idea, BiYacc programs can also be made more concise: In a BiYacc program, the user is allowed to omit the production rules in the concrete syntax part (or omit the whole concrete syntax part), and they will be automatically generated by extracting the terminals and nonterminals in the right-hand sides of all actions. However, if these production rules are supplied, BiYacc will perform some sanity checks: It will make sure that, in an action group, the user has covered all of the production rules of the nonterminal appearing in the 'type declaration', and never uses undefined production rules.

Just like basic BiYacc, all of the systems described above (aim to) handle unambiguous grammars only. Theoretically, when the user-defined grammar (or the derived grammar) is ambiguous, ISDs' partial isomorphism could guarantee Theorem 1′ by returning Nothing on ambiguous input; FliPpr's (own) Theorem 1 is comparable to Theorem 1′ by taking all the language constructs which may cause non-injective printing into account. However, according to the paper, FliPpr's Theorem 1 appears to only consider nondeterministic printing based on prettiness (layouts). Since

the discussion on ambiguous grammars has not been presented in their papers, we also tested their implementation and the behaviour is as follows: Neither ISDs nor FliPpr will notify the user that the (derived) grammar is ambiguous at compile time. For ISDs, the right-to-left direction of our Theorem 1′ will fail, while for FliPpr, both directions will fail. (They never promise to handle ambiguous grammars, though.) In contrast, Brabrand et al. give a detailed discussion about ambiguity detection, and XSugar statically checks if the transformations are 'reversible'. If any ambiguity in the program is detected, XSugar will notify the user of the precise location where ambiguity arises. In BiYacc, the ambiguity detection of the input grammar is performed by the employed parser generator (currently Happy), and the result is reported at compile time; if no warning is reported, the well-behavedness is always guaranteed. Note that the ambiguity detection can produce false positives: warnings only mean that the grammar is not LALR(1) but does not necessarily mean that the grammar is ambiguous—ambiguity detection is undecidable for the full CFG (Cantor, 1962).

Finally, we compare BiYacc with an industrial tool, Augeas, which provides the user with a local configuration API that converts configuration data into a rose tree representation (Lutterkort, 2008). Similar to BiYacc, Augeas also uses the idea of state-based asymmetric lenses so that its *parse* and *print* functions satisfy well-behavedness and it tries to preserve comments and layouts when printing the tree representation back. However, since the purpose of Augeas and BiYacc is different, the differences between the tools are also noticeable: (i) Augeas works for regular grammars while BiYacc works for (unambiguous) context-free grammars. (ii) Augeas uses a combinator-based approach while BiYacc adopts a generator-based approach. (iii) Augeas works more like a simple parser that stops after constructing CSTs: in the parsing direction, Augeas unambiguously separates strings into sub-strings, turn sub-strings into tokens, and use tokens to build the corresponding tree; but since each lens combinator (of Augeas) has its predefined strategy to turn its acceptable strings into the tree representation, the corresponding (rose) tree will be determined once the input string and the lens combinators for parsing the string are given; Augeas does not provide a functionality to further transform a rose tree. On the other hand, BiYacc first turns a string into its isomorphic CST (fully determined the input string and the grammar description) and finally converts the CST to its AST in accordance with the algebraic data types defined by the user; that is, the relation between a string (CST) and its AST is not predetermined but can be adjusted by the user (through printing actions).

3.4.2 Comparison with a Get-based Approach

As explained in Section 2.3.3, the purpose of bidirectional programming is to relieve the burden of thinking bidirectionally—the programmer writes a program in only one direction, and a program in the other direction is derived automatically. In the context of parsing and retentive printing, the get-based approach lets the programmer describe a parser, whereas the put-based approach lets the programmer describe a printer. Below we discuss in more depth how the put-based methodology affects BiYacc's design by comparing BiYacc with a closely related, get-based system.

Martins et al. (2014) introduces an attribute grammar–based BX system for defining transformations between two representations of languages (two grammars). The utilisation is similar to BiYacc: The programmer defines both grammars and a set of rules specifying a *forward* transformation (i.e. *get*), with a backward transformation (i.e. *put*) being automatically generated. For example, the BiYacc actions in lines 28–30 of Figure 3.1 can be expressed in Martins et al.'s system as

$$\begin{aligned} get_A^E(plus \quad (x, \text{`+'}, y)) &\rightarrow add(get_A^E(x), get_A^T(y)) \\ get_A^E(minus(x, \text{`-'}, y)) &\rightarrow sub(get_A^E(x), get_A^T(y)) \\ get_A^E(fromt(e)) &\rightarrow get_A^T(e) \end{aligned}$$

which describes how to convert certain forms of CSTs to corresponding ASTs. The similarity is evident, and raises the question as to how get-based and put-based approaches differ in the context of parsing and retentive printing.

The difference lies in the fact that, with a get-based system, certain decisions on the backward transformation are, by design, permanently encoded in the bidirectionalisation system and cannot be controlled by the user, whereas a put-based system can give the user fuller control. For instance, when no source is given and more than one rules can be applied, Martins et al.'s system chooses, by design, the one that creates the most specialised version. This might or might not be ideal for the user of the system. As an example, suppose that we port to Martins et al.'s system the BiYacc action that relates negation with an abstract `Sub` expression, coexisting with a more general rule that maps a concrete subtraction to an abstract `Sub` expression. Then printing the AST `Sub (Num 0) (Var "a")` from scratch will and can only produce `-a`, as dictated by the system's hard-wired printing logic. By contrast, the user of BiYacc can easily choose to print the AST from scratch as `-a` or `0 - a` by suitably ordering the actions.

This difference is somewhat subtle, and one might argue that Martins et al.'s design simply went one step too far—if their system had been designed to respect the rule ordering as specified by the user, as opposed to always choosing the most specialised rule, the system would have given its

user the same flexibility as BiYacc. Interestingly, whether to let user-specified rule/action ordering affect the system's behaviour is, in this case, exactly the line between get-based and put-based design. The user of Martins et al.'s system writes rules to specify a *get* transformation, whose semantics is the same regardless of how the rules are ordered, and thus it would be unpleasantly surprising if the rule ordering turned out to affect the system's behaviour. By contrast, the user of BiYacc only needs to think in one direction about the printing behaviour, for which it is natural to consider how the actions should be ordered when an AST has many corresponding CSTs; the parsing behaviour will then be automatically and uniquely determined. In short, relevance of action ordering is incompatible with get-based design, but is a natural consequence of put-based thinking.

4

Bidirectionalised Filters for Handling Grammatical Ambiguity

In this chapter, we make an extension to our solution (mainly) for the isomorphism part (between program text and CSTs). In Chapter 3, we have described the basic version of BiYacc, about which there is an important assumption (stated in Section 3.2.2.3) that grammars have to be unambiguous. Having this assumption can be rather inconvenient in practice, however, as ambiguous grammars (with disambiguation directives) are often preferred since they are considered more natural and human-friendly than their unambiguous versions (Afroozeh and Izmaylova, 2015; Klint and Visser, 1994). Therefore, the purpose of this section is to revise the architecture of basic BiYacc to allow the use of ambiguous grammars and disambiguation directives. This is in fact a long-standing problem, for tools designed for building (consistent) parser and printer pairs usually do not support such functionality (Section 3.4).

For example, consider the ambiguous grammar (with disambiguation directives) and printing actions in Figure 4.1, which we will refer to throughout this section. Note that the parenthesis structure is dropped when converting a CST to its AST (as stated by the last printing action of `Arith +> Expr`). The grammar is converted to CST datatypes and constructors as in Section 3.2.2.1,

```
#Concrete                                        #Directives
Expr -> [Plus]       Expr '+' Expr               Priority:
      | [Minus]      Expr '-' Expr               Times > Plus     ;
      | [Times]      Expr '*' Expr               Times > Minus    ;
      | [Division]   Expr '/' Expr               Division > Plus  ;
      | [Paren]      '(' Expr ')'                Division > Minus ;
      | [Lit]        Numeric
      ;                                          Associativity:
                                                 Left: Plus, Minus, Times, Division ;

#Actions
Arith +> Expr
  Add x y  +>  [x +> Expr] '+' [y +> Expr] ;
  Sub x y  +>  [x +> Expr] '-' [y +> Expr] ;
  Mul x y  +>  [x +> Expr] '*' [y +> Expr] ;
  Div x y  +>  [x +> Expr] '/' [y +> Expr] ;
  Num i    +>  [i +> Numeric]              ;
  e        +>  '(' [e +> Expr] ')'         ;
;;
```

Figure 4.1: Arithmetic expressions defined by an ambiguous grammar and the corresponding printing actions. For simplicity, the variable and negation productions are omitted.

but here we explicitly give names such as `Plus` and `Times` to production rules, and these names (instead of automatically generated ones) are used for constructors in CSTs. Compared with this grammar, the unambiguous one shown in Figure 3.1 is less intuitive as it uses different nonterminals to resolve the ambiguity regarding operator precedence and associativity.

In this chapter, we explain the problem brought by ambiguous grammars (Section 4.1) and address it (Section 4.2) using *generalised parsing* and *bidirectionalised filters* (bi-filters for short). Then we extend BiYacc with bi-filters (Section 4.3) while still respecting consistency of the generated parser and printer pairs. To program with bi-filters easily, we provide compositional bi-filter directives (Section 4.4) which compile to priority and associativity bi-filters; power users can also define their own bi-filters (Section 4.5) and we illustrate this by writing a bi-filter that solves the (in)famous dangling-else problem. With bi-filters, we rewrite the Tiger language in its ambiguous form (that is closer to its original definition) and disambiguate it; we highlight the changes in the grammar and in the case study compared to the unambiguous one (Section 4.6). Finally, we present related work regarding generalised parsing and disambiguation filters (Section 4.7) and discuss several issues (Section 4.8) regarding implementation details and future research directions.

4.1 Problems with Ambiguous Grammars

Consider the original architecture of BiYacc in Figure 3.2, which we want to (and basically will) retain while adapting it to support ambiguous grammars. The first component (of the executable) we should adapt is *cparse* :: String → Maybe CST, the (concrete) parsing direction of the isomorphism: since there can be multiple CSTs corresponding to the same program text, *cparse* needs to choose one of them as the result. Disambiguation directives (Johnson, 1975) were invented to describe how to make this choice. For example, with respect to the grammar in Figure 4.1, text `1 + 2 * 3` will have either of the two CSTs[1]:

$$\texttt{cst}_1 = \sharp\texttt{Plus 1 (Times 2 3)}$$
$$\texttt{cst}_2 = \sharp\texttt{Times (Plus 1 2) 3}$$

depending on the precedence of addition and multiplication. Conventionally, we can use the Yacc-style disambiguation directives `%left '+'; %left '*';` to specify that multiplication has higher precedence over addition, and instruct the parser to choose $\texttt{cst}_1$.

However, merely adapting *cparse* with disambiguation behaviour is not enough, since the isomorphism (Theorem 3.2.2), in particular its right to left direction (which is simplified as *cparse* (*cprint cst*) = Just *cst*), cannot be established when an ambiguous grammar is used—in the example above, $\texttt{cparse (cprint cst}_2\texttt{) = Just cst}_1 \neq \texttt{Just cst}_2$. This is because the image of *cparse* is strictly smaller than the domain of *cprint* (in general): if we start from any CST not in the image of *cparse*, we will never be able to get back to the same CST through *cprint* and then *cparse*. This tells us that, to retain the isomorphism, the domain of *cprint* should not be the whole CST but only the image of *cparse*, i.e. the set of *valid* CSTs (as defined by the disambiguation directives), which we denote by CST_F (for reasons that will be made clear in Section 4.3).

Now that the right-hand side domain of the isomorphism is restricted to CST_F, the source of the lens should be restricted to this set as well. For *get* :: CST → Maybe AST we need to restrict its domain, which is easy; for *put* :: CST → AST → Maybe CST we should revise its type to CST_F → AST → Maybe CST_F, meaning that *put* should now guarantee that the CSTs it produces are valid, which is nontrivial. For example, consider the result of `put cst ast` where `ast = Mul (Add (Num 1) (Num 2)) (Num 3)` and `cst` is some arbitrary tree. A natural choice is $\texttt{cst}_2$, which, however, is excluded from CST_F by disambiguation. A possible solution could be making *put* refuse to produce a result from `ast`, but this is unsatisfactory since `ast` is perfectly valid and should not be ignored by *put*. A more satisfactory way is to create a CST with *proper parentheses*, like

[1] For simplicity, we use $\sharp$ to annotate type-incorrect CSTs in which fields for layouts (and comments) and unimportant constructors such as `Lit` are omitted.

$\texttt{cst}_3$ = ♯`Times (Paren (Plus 1 2)) 3`. But it is not clear in what cases parentheses need to be added, in what cases they need not, and in what cases they cannot.

We are now led to a fundamental problem: generally, *put* strategies for producing valid CSTs should be inferred from the disambiguation directives, but the semantics of Yacc disambiguation directives are defined over the implementation of Yacc's underlying LR parsing algorithm with a stack (Aho et al., 1975; Johnson, 1975), and therefore it is nontrivial to invent a dual semantics in the *put* direction. To have a simple and clear semantics of the disambiguation process, we turn away from Yacc's traditional approach and opt for an alternative approach based on *generalised parsing* with *disambiguation filters* (Klint and Visser, 1994; van den Brand et al., 2002), whose semantics can be specified implementation-independently. Based on this simple and clear semantics, we will be able to devise ways to amend *put* to produce only valid CSTs, and formally state the conditions under which the executable generated by the revised BiYacc is well-behaved.

4.2 Generalised Parsing and Bidirectionalised Filters

The idea of generalised parsing is for a parser to produce all possible CSTs corresponding to its input program text instead of choosing only one CST (possibly prematurely) (Earley, 1970; Scott and Johnstone, 2010; Tomita, 1985; Younger, 1967), and works naturally with ambiguous grammars. In practice, a generalised parser can be generated using, e.g., Happy's GLR mode (Marlow and Gill, 2001), and we will assume that given a grammar we can obtain a generalised parser:

$$\mathit{cgparse} :: \mathsf{String} \to [\mathsf{CST}] \,.$$

The result of *cgparse* is a list of CSTs. We do not need to wrap the result type in Maybe—if *cgparse* fails, an empty list is returned. And we should note that, while the result is a list, what we really mean is a set (commonly represented as a list in Haskell) since we do not care about the order of the output CSTs and do not allow duplicates.

With generalised parsing, program text is first parsed to all the possible CSTs; disambiguation then becomes an extremely simple concept: removing CSTs that the user does not want. One possible semantics of disambiguation may be a function *judge* :: Tree → Bool; during disambiguation, this function is applied to all candidate CSTs, and a candidate *cst* is removed if *judge cst* returns False, or kept otherwise. We call these functions *disambiguation filters* ('filters' for short). [1] For

[1] The general type for disambiguation filters is $[t] \to [t]$, which allows comparison among a list of CSTs. However, since in this paper we only consider *property filters* defined in terms of predicates (on a single tree), it is sufficient to use

example, to state that top-level addition is left-associative, we can use the following filter[1] to reject right-sided trees:

```
plusJudge :: Expr -> Bool
plusJudge (♯Plus _ (Plus _ _)) = False
plusJudge _ = True .
```

This simple and clean semantics of disambiguation is then amenable to 'bidirectionalisation', which we do next.

Note that, unlike YACC's disambiguation directives, which assign precedence and associativity to individual tokens and *implicitly* exclude 'some' CSTs, in `plusJudge` above we *explicitly* ban incorrect CSTs through pattern matching. Having described which CSTs are incorrect, we can further specify what to do with incorrect CSTs in the printing direction. Whenever a CST 'in a bad shape', i.e. rejected by a filter like `plusJudge`, is produced, we can repair it so that it becomes 'in a good shape':

```
plusRepair :: Expr -> Expr
plusRepair (♯Plus t1 (Plus t2 t3)) = ♯Plus t1 (Paren (Plus t2 t3))
plusRepair t = t .
```

The above function states that whenever a `Plus` is another `Plus`'s right child, there must be a parenthesis structure `Paren` in between. Observant readers might have found that the trees processed by `plusJudge` and `plusRepair` have the same pattern. We can therefore pair the two functions and make a *bidirectionalised filter* ('bi-filters' for short):

```
plusLAssoc :: Expr -> (Expr, Bool)
plusLAssoc (♯Plus t1 (Plus t2 t3)) = (♯Plus t1 (Paren (Plus t2 t3)), False)
plusLAssoc t = (t, True) .
```

But there is still some redundancy in the definition of `plusLAssoc`, for when the input tree is correct we always return the same input tree; this can be further optimised:

```
plusLAssoc' :: Expr -> Maybe Expr
plusLAssoc' (♯Plus t1 (Plus t2 t3)) = Just (♯Plus t1 (Paren (Plus t2 t3)))
plusLAssoc' _ = Nothing .
```

the simplified type $t \to \text{Bool}$.

[1] This is not a very realistic filter, although it sufficiently demonstrates the use of filters and removes ambiguity in simplest cases like `1 + 2 * 3`. In general, the filter should be *complete* (Definition 4.3.2) so that ambiguity is fully removed from the grammar.

Generalising the example above, we arrive at the definition of bi-filters.

Definition 4.2.1 (Bidirectionalised Filters)**.** A *bidirectionalised filter* F working on trees of type t is a function of type BiFilter t defined by

$$\textbf{type}\ \mathsf{BiFilter}\ t\ =\ t \to \mathsf{Maybe}\ t$$

satisfying

$$\mathit{repair}\ F\ t = t' \quad \Rightarrow \quad \mathit{judge}\ F\ t' = \mathsf{True} \tag{RepairJudge}$$

where the two directions *repair* and *judge* are defined by

$$\begin{aligned}
&\mathit{repair} :: \mathsf{BiFilter}\ t \to (t \to t)\\
&\mathit{repair}\ F\ t = \textbf{case}\ F\ t\ \textbf{of}\\
&\qquad\qquad\qquad \mathsf{Nothing} \to t\\
&\qquad\qquad\qquad \mathsf{Just}\ t' \quad \to t'\\
&\mathit{judge} :: \mathsf{BiFilter}\ t \to (t \to \mathsf{Bool})\\
&\mathit{judge}\ F\ t = \textbf{case}\ F\ t\ \textbf{of}\\
&\qquad\qquad\qquad \mathsf{Nothing} \to \mathsf{True}\\
&\qquad\qquad\qquad \mathsf{Just}\ _ \quad \to \mathsf{False}\ .
\end{aligned}$$

The functions *repair* and *judge* accept a bi-filter and return respectively the specialised *repair* and *judge* functions for that bi-filter. For clarity, we let repair_F denote *repair* F and let judge_F denote *judge* F. The bi-filter law RepairJudge dictates that repair_F should bring its input tree into a correct state with respect to judge_F. The reader may wonder why there is not a dual JudgeRepair law saying that if a tree is already of an allowed form justified by judge_F, then repair_F should leave it unchanged. In fact this is always satisfied according to the definitions of *judge* and *repair*, so we formulate it as a lemma.

Lemma 4.2.2 (JudgeRepair)**.** Any bi-filter F satisfies the *JudgeRepair* property:

$$\mathit{judge}_F\ t = \mathsf{True} \quad \Rightarrow \quad \mathit{repair}_F\ t = t\ .$$

Proof. From $\mathit{judge}_F\ t = \mathsf{True}$ we deduce $F\ t = \mathsf{Nothing}$, which implies $\mathit{repair}_F\ t = t$. □

In the next section, we will describe how to fit generalised parsers and bi-filters into the architecture of BiYacc. To let bi-filters work with the lens part between CSTs and ASTs, we require

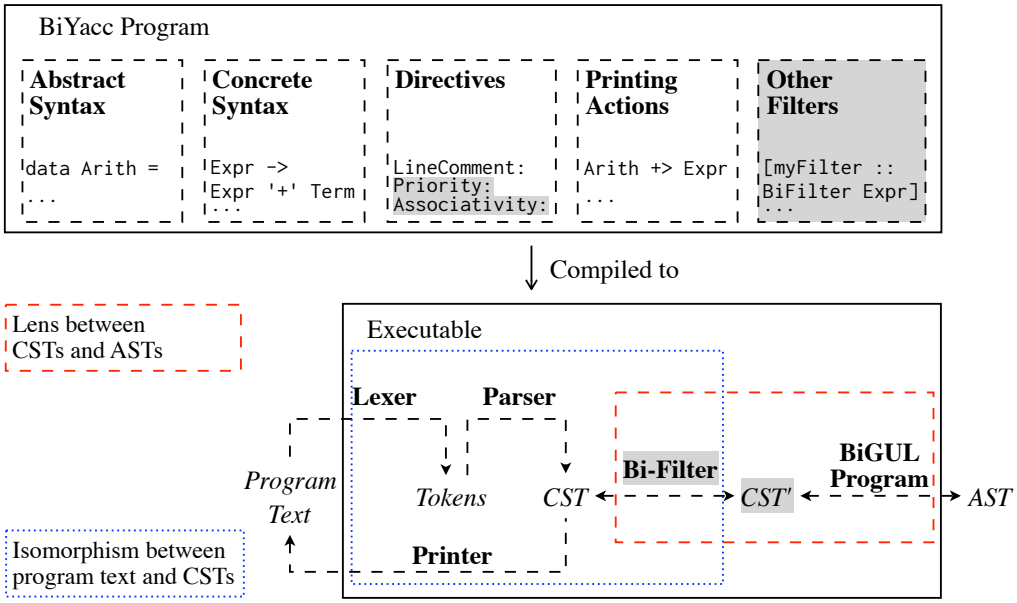


Figure 4.2: New architecture of BiYacc. (New components are in light grey.)

a further property characterising the interaction between the repairing direction of a bi-filter and the get direction of a lens.

Definition 4.2.3 (PassThrough)**.** A bi-filter F satisfies the *PassThrough* property with respect to a function *get* exactly when

$$get \circ repair_F = get\ .$$

If we think of a *get* function as mapping CSTs to their semantics (in our case ASTs), then the PassThrough property is a reasonable requirement since it guarantees that the repaired CST will have the same semantics as before (as it is converted to the same AST). This property will be essential for establishing the well-behavedness of the executable generated by the revised BiYacc.

4.3 The New BiYacc System for Ambiguous Grammars

As depicted in Figure 4.2, the executable generated by the new BiYacc system is still the composition of an isomorphism and a lens, which is the structure we have tried to retain. To precisely identify the changes in several generated components (in the executable file) and demonstrate how parsing and printing work with a bi-filter, we present Figure 4.3 and will use this one instead. In the new system, we will still use the *get* and *put* transformations generated from printing actions and

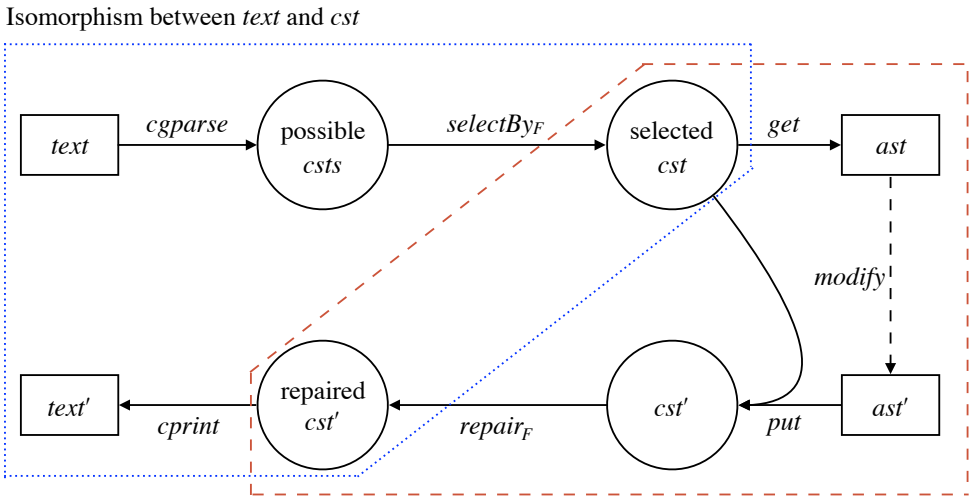


Figure 4.3: A schematic diagram showing how parsing and printing work with a bi-filter.

the concrete printer *cprint* from grammars, while the concrete parser *cparse* is replaced with a generalised parser *cgparse*. Additionally, the `#Directives` and `#OtherFilters` parts will be used to generate a bi-filter F, whose $judge_F$ (used in the $selectBy_F$ function in Figure 4.3) and $repair_F$ components are integrated into the isomorphism and lens parts respectively, so that the right-hand side domain of the isomorphism and the source of the lens become CST_F, the set of valid CSTs:

$$\mathsf{CST}_F = \{\, cst \in \mathsf{CST} \mid judge_F\ cst = \mathsf{True} \,\}\ .$$

Next, we introduce the (new) isomorphism and lens parts, and prove their inverse properties and well-behavedness respectively.

4.3.1 The Revised Isomorphism between Program Text and CSTs

Let us first consider the isomorphism part between String and CST_F, which is enclosed within the blue dotted lines in Figure 4.3 and consists of *cprint*, *cgparse*, and $selectBy_F$:

$$\begin{aligned}
&cprint :: \mathsf{CST} \to \mathsf{String}\\
&cgparse :: \mathsf{String} \to [\mathsf{CST}]\\
&selectBy_F :: [\mathsf{CST}] \to \mathsf{Maybe}\ \mathsf{CST}_F\\
&selectBy_F\ csts = \mathbf{case}\ selectBy\ judge_F\ csts\ \mathbf{of}\\
&\qquad\qquad [cst]\ to\ \mathsf{Just}\ cst
\end{aligned}$$

$$
\begin{aligned}
&\_ \quad \rightarrow \mathsf{Nothing} \\
&\mathit{selectBy} :: (a \rightarrow \mathsf{Bool}) \rightarrow [a] \rightarrow [a] \\
&\mathit{selectBy}\ p\ [\,] = [\,] \\
&\mathit{selectBy}\ p\ (x : xs) \mid p\ x = x : \mathit{selectBy}\ p\ xs \\
&\mathit{selectBy}\ p\ (x : xs) \mid \mathit{otherwise} = \mathit{selectBy}\ p\ xs\ .
\end{aligned}
$$

In the parsing direction, first *cgparse* produces all the CSTs; then $\mathit{selectBy}_F$ utilises a function *selectBy* and a predicate judge_F to (try to) select the only correct *cst*; if there is no correct CST or more than one correct CST, Nothing is returned. The function *selectBy*, which selects from the input list exactly the elements satisfying the given predicate, is named *filter* in Haskell's standard libraries but renamed here to avoid confusion. In the printing direction, we still use *cprint* to flatten a (correct) CST back to program text. Formally, constructed from *cgparse* and *cprint*, the two directions of the isomorphism are

$$
\begin{aligned}
\mathit{cparse}_F &:: \mathsf{String} \rightarrow \mathsf{Maybe}\ \mathsf{CST}_F \\
\mathit{cparse}_F &= \mathit{selectBy}_F \circ \mathit{cgparse} \\
\mathit{cprint}_F &:: \mathsf{CST}_F \rightarrow \mathsf{Maybe}\ \mathsf{String} \\
\mathit{cprint}_F &= \mathsf{Just} \circ \mathit{cprint}\ .
\end{aligned}
$$

We are eager to give the revised version of the inverse properties (Theorem 4.3.5) and their proofs, which, however, depend on two assumptions about generalised parsers and bi-filters. So let us present them in order.

Definition 4.3.1 (Generalised Parser Correctness)**.** A generalised parser *cgparse* is *correct* with respect to a printer *cprint* exactly when

$$\mathit{cgparse}\ \mathit{text} = \{\ \mathit{cst} \in \mathsf{CST} \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \}\ .$$

This is exactly Definition 3.7 of Klint and Visser (1994). We remind the reader again that we use sets and lists interchangeably for the parsing results.

Definition 4.3.2 (Bi-Filter Completeness)**.** A bi-filter F is *complete* with respect to a printer *cprint* exactly when

$$\mathit{text} \in \mathsf{Img}\ \mathit{cprint} \quad \Rightarrow \quad \left|\{\ \mathit{cst} \in \mathsf{CST}_F \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \}\right| = 1\ .$$

($\mathsf{Img}\ f = \{\ y \mid \exists x.\ f\,x = y\ \}$ is the image of the function f.)

This is revised from Definition 4.3 of Klint and Visser (1994), where they require that filters select exactly one CST and reject all the others. Since it is undecidable to judge whether a given context-free grammar is ambiguous (Cantor, 1962), we cannot tell whether a (bi-)filter (for the full CFG) is complete, either. But still, some checks can be performed on simple cases, as stated in Section 4.7.

The following two lemmas connect our two assumptions, Definitions 4.3.1 and 4.3.2, with the definitions of $cparse_F$ and $cprint_F$.

Lemma 4.3.3. Given $cparse_F$ and $cprint_F$ where *cgparse* is correct and F is complete with respect to *cprint*, we have

$$text \in \mathsf{Img}\ cprint \quad \Rightarrow \quad \exists\, cst \in \mathsf{CST}_F.\ cparse_F\ text = \mathsf{Just}\ cst\ \wedge\ cprint\ cst = text\ .$$

Proof. We reason:

$$\begin{aligned}
& selectBy_F\ (cgparse\ text) \\
= \quad & \{\ \text{Definition of } SelectBy_F\ \} \\
& \textbf{case}\ selectBy\ judge_F\ (cgparse\ text)\ \textbf{of}\ \{\ [cst] \to \mathsf{Just}\ cst; _ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Generalised Parser Correctness}\ \} \\
& \textbf{case}\ selectBy\ judge_F\ \{\ cst \in \mathsf{CST} \mid cprint\ cst = text\ \}\ \textbf{of} \\
& \qquad \{\ [cst] \to \mathsf{Just}\ cst;\ _ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ selectBy\ judge_F \text{ only selects correct CSTs regarding } F\ \} \\
& \textbf{case}\ \{\ cst \in \mathsf{CST}_F \mid cprint\ cst = text\ \}\ \textbf{of}\ \{\ [cst] \to \mathsf{Just}\ cst;\ _ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Bi-Filter Completeness}, \exists cst' \text{ s.t. } \{\ cst \in \mathsf{CST}_F \mid cprint\ cst = text\ \} = [cst']\ \} \\
& \textbf{case}\ [cst']\ \textbf{of}\ \{\ [cst] \to \mathsf{Just}\ cst;\ _ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Definition of } \textbf{case}\ \} \\
& \mathsf{Just}\ cst\ .
\end{aligned}$$

Moreover, *cst* satisfies $cprint\ cst = text$, since the latter is the comprehension condition of the set from which *cst* is chosen, and therefore $cprint_F\ cst = \mathsf{Just}\ text$. □

Lemma 4.3.4 (Printer Injectivity)**.** If F is a complete bi-filter, then $cprint_F$ is injective.

Proof. Assume that $cst, cst' \in \mathsf{CST}_F$ and $cprint\ cst = cprint\ cst' = text$ for some $text$; that is, both cst and cst' are in the set $P = \{\, cst \in \mathsf{CST}_F \mid cprint\ cst = text \,\}$. Since $text \in \mathsf{Img}\ cprint$, by the completeness of F we have $P = 1$, and hence $cst = cst'$. □

We can now prove a generalised version of Theorem 3.2.2 for ambiguous grammars.

Theorem 4.3.5 (Inverse Properties with Bi-Filters)**.** Given $cparse_F$ and $cprint_F$ where $cgparse$ is correct and F is complete, then the following holds:

$$cparse_F\ text = \mathsf{Just}\ cst \quad \Rightarrow \quad cprint_F\ cst = \mathsf{Just}\ text \tag{4.1}$$

$$cprint_F\ cst = \mathsf{Just}\ text \quad \Rightarrow \quad cparse_F\ text = \mathsf{Just}\ cst\ . \tag{4.2}$$

Proof. For (4.1): Let $\mathsf{Just}\ cst = selectBy_F\ (cgparse\ text)$. According to the definition of $selectBy_F$, we have $cst \in cgparse\ text$. By Generalised Parser Correctness $cprint\ cst = text$, and therefore $cprint_F\ cst = \mathsf{Just}\ text$.

For (4.2): The antecedent implies $cprint\ cst = text$. By Lemma 4.3.3, we have $cparse_F\ text = \mathsf{Just}\ cst'$ for some $cst' \in \mathsf{CST}_F$ such that $cprint_F\ cst' = \mathsf{Just}\ text = cprint_F\ cst$. By Lemma 4.3.4 we know $cst' = cst$, and thus $cparse_F\ text = \mathsf{Just}\ cst$. □

4.3.2 The Revised Lens between CSTs and ASTs

Recall that the `#Action` part of a BiYacc program produces a lens (BiGUL program) consisting of a pair of well-behaved *get* and *put* functions:

$$\begin{aligned} &get :: \mathsf{CST} \to \mathsf{Maybe\ AST} \\ &put :: \mathsf{CST} \times \mathsf{AST} \to \mathsf{Maybe\ CST}\ . \end{aligned}$$

To work with a bi-filter F, in particular its $repair_F$ component, they need to be adapted to get_F and put_F, which accept only valid CSTs:

$$\begin{aligned} get_F &:: \mathsf{CST}_F \to \mathsf{Maybe\ AST} \\ get_F &= get \\ put_F &:: \mathsf{CST}_F \times \mathsf{AST} \to \mathsf{Maybe\ CST}_F \\ put_F\ (cst,\ ast) &= fmap\ repair_F\ (put\ (cst,\ ast)) \end{aligned}$$

where *fmap* is a standard HASKELL library function defined (for Maybe) by

$$\begin{array}{lcl} \mathit{fmap} & :: & (a \to b) \to \mathsf{Maybe}\ a \to \mathsf{Maybe}\ b \\ \mathit{fmap}\ f\ \mathsf{Nothing} & = & \mathsf{Nothing} \\ \mathit{fmap}\ f\ (\mathsf{Just}\ x) & = & \mathsf{Just}\ (f\ x)\ . \end{array}$$

We will need a lemma about *fmap*, which can be straightforwardly proved by a case analysis.

Lemma 4.3.6. If $\mathit{fmap}\ f\ \ mx = \mathsf{Just}\ y$, then there exists x such that $mx = \mathsf{Just}\ x$ and $f\ x = y$.

Now we prove that get_F and put_F are well-behaved, which is a generalisation of Theorem 3.2.3 for ambiguous grammars.

Theorem 4.3.7 (Well-behavedness with Bi-Filters)**.** Given a complete bi-filter F and a well-behaved lens consisting of *get* and *put*, if *get* and F additionally satisfy PassThrough, then the get_F and put_F functions with respect to F are also well-behaved:

$$\mathit{put}_F\ (\mathit{cst},\ \mathit{ast}) = \mathsf{Just}\ \mathit{cst} \quad \Rightarrow \quad \mathit{get}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{ast} \tag{4.3}$$

$$\mathit{get}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{ast} \quad \Rightarrow \quad \mathit{put}_F\ (\mathit{cst},\ \mathit{ast}) = \mathsf{Just}\ \mathit{cst}\ . \tag{4.4}$$

Proof. For (4.3): The antecedent expands to $\mathit{fmap}\ \mathit{repair}_F\ (\mathit{put}\ (\mathit{cst},\ \mathit{ast})) = \mathsf{Just}\ \mathit{cst}$, which, by Lemma 4.3.6, implies $\mathit{put}\ (\mathit{cst},\ \mathit{ast}) = \mathsf{Just}\ \mathit{cst}''$ for some cst'' such that $\mathit{repair}_F\ \mathit{cst}'' = \mathit{cst}'$. Now we reason:

$$\begin{array}{ll} & \mathit{get}_F\ \mathit{cst}' \\ = & \quad \{\ \text{Definition of } \mathit{get}_F \text{ and } \mathit{cst} \in \mathsf{CST}_F\ \} \\ & \mathit{get}\ \mathit{cst}' \\ = & \quad \{\ \text{Definition of } \mathit{cst}'\ \} \\ & \mathit{get}\ (\mathit{repair}_F\ \mathit{cst}'') \\ = & \quad \{\ \text{PassThrough}\ \} \\ & \mathit{get}\ \mathit{cst}'' \\ = & \quad \{\ \text{Correctness}\ \} \\ & \mathsf{Just}\ \mathit{ast}\ . \end{array}$$

For (4.4):

$$
\begin{aligned}
& put_F\ (cst,\ ast) \\
= \quad & \{\ \text{Definition of } put_F\ \} \\
& fmap\ repair_F\ (put\ (cst,\ ast)) \\
= \quad & \{\ \text{Hippocraticness}\ \} \\
& fmap\ repair_F\ (\mathsf{Just}\ cst) \\
= \quad & \{\ \text{Definition of } fmap\ \} \\
& \mathsf{Just}\ (repair_F\ cst) \\
= \quad & \{\ \text{Since } cst \in \mathsf{CST}_F,\ judge_F\ cst = \mathsf{True}.\ \text{By JudgeRepair}\ \} \\
& \mathsf{Just}\ cst\ .
\end{aligned}
$$

□

4.4 Bi-Filter Directives

Until now, we have only considered working with a single bi-filter, but this is without loss of generality because we can provide a bi-filter composition operator (Section 4.4.1) so that we can build large bi-filters from small ones. This is a suitable semantic foundation for introducing YAcc-like directives for specifying priority and associativity into BiYAcc (Section 4.4.2), since we can give these directives a bi-filter semantics and interpret a collection of directives as the composition of their corresponding bi-filters. We will also discuss some properties related to this composition (Section 4.4.3).

4.4.1 Bi-Filter Composition

We start by defining bi-filter composition, with the intention of making the net effect of applying a sequence of bi-filters one by one the same as applying their composite. Although the intention is better captured by Lemma 4.4.2, which describes the *repair* and *judge* behaviour of a composite bi-filter in terms of the component bi-filters, we give the definition of bi-filter composition first.

Definition 4.4.1 (Bi-Filter Composition)**.** The composition of two bi-filters is defined by

$$
\begin{aligned}
& (\triangleleft) :: (t \to \mathsf{Maybe}\ t)\ \to\ (t \to \mathsf{Maybe}\ t)\ \to\ (t \to \mathsf{Maybe}\ t) \\
& (j \triangleleft i)\ t = \textbf{case}\ i\ t\ \textbf{of}
\end{aligned}
$$

$$
\begin{array}{l}
\mathsf{Nothing} \longrightarrow j\ t \\
\mathsf{Just}\ t' \quad \longrightarrow \textbf{case}\ j\ t'\ \textbf{of} \\
\quad \mathsf{Nothing} \longrightarrow \mathsf{Just}\ t' \\
\quad \mathsf{Just}\ t'' \quad \longrightarrow \mathsf{Just}\ t''\ .
\end{array}
$$

When applying a composite bi-filter $j \triangleleft i$ to a tree t, if t is correct with respect to i (i.e. $i\ t = \mathsf{Nothing}$), we directly pass the original tree t to j; otherwise t is repaired by i, yielding t', and we continue to use j to repair t'. Note that if $j\ t' = \mathsf{Nothing}$, we return the tree t' instead of Nothing.

Lemma 4.4.2. For a composite bi-filter $j \triangleleft i$, the following two equations hold:

$$
\begin{aligned}
&\mathit{repair}\ (j \triangleleft i)\ t = (\mathit{repair}_j \circ \mathit{repair}_i)\ t \\
&\mathit{judge}\ (j \triangleleft i)\ t = \mathit{judge}_j\ t \wedge \mathit{judge}_i\ t\ .
\end{aligned}
$$

Proof. By the definition of bi-filter composition. □

Composition of bi-filters should still be a bi-filter and satisfy RepairJudge and PassThrough. This is not always the case though—to achieve this, we need some additional constraint on the component bi-filters, as formulated below.

Definition 4.4.3. Let i and j be bi-filters. We say that j *respects* i exactly when

$$
\mathit{judge}_i\ t = \mathsf{True} \quad \Rightarrow \quad \mathit{judge}_i\ (\mathit{repair}_j\ t) = \mathsf{True}\ .
$$

If j respects i, then a later applied repair_j will never break what may already be repaired by a previous repair_i. Thus in this case we can safely compose a j after i. This is proved as the following theorem.

Theorem 4.4.4. Let i and j be bi-filters (satisfying RepairJudge and PassThrough). If j respects i, then $j \triangleleft i$ also satisfy RepairJudge and PassThrough.

Proof. For RepairJudge, we reason:

$$
\begin{aligned}
& \mathit{judge}\ (j \triangleleft i)\ (\mathit{repair}\ (j \triangleleft i)\ t) \\
= \quad & \{\ \text{Lemma 4.4.2}\ \} \\
& \mathit{judge}\ (j \triangleleft i)\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\
= \quad & \{\ \text{Lemma 4.4.2}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{judge}_j\ (\mathit{repair}_j\ (\mathit{repair}_i\ t))\ \wedge\ \mathit{judge}_i\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\
= \quad & \{\ \text{RepairJudge of } j\ \} \\
& \mathsf{True}\ \wedge\ \mathit{judge}_i\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\
= \quad & \{\ \mathit{judge}_i\ (\mathit{repair}_i\ t') = \mathsf{True};\ j \text{ respects } i\ \} \\
& \mathsf{True}\ \wedge\ \mathsf{True} \\
= \ & \mathsf{True}\ .
\end{aligned}
$$

And for PassThrough:

$$
\begin{aligned}
& \mathit{get}\ (\mathit{repair}\ (j \triangleleft i)\ t) \\
= \quad & \{\ \text{Lemma 4.4.2}\ \} \\
& \mathit{get}\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\
= \quad & \{\ \text{PassThrough of } j\ \} \\
& \mathit{get}\ (\mathit{repair}_i\ t) \\
= \quad & \{\ \text{PassThrough of } i\ \} \\
& \mathit{get}\ t\ .
\end{aligned}
$$

□

4.4.2 Priority and Associativity Directives

To relieve the burden of writing bi-filters manually and guaranteeing respect among bi-filters being composed, we provide some directives for constructing bi-filters dealing with priority[1] and associativity, which are generally comparable to YACC's conventional disambiguation directives. The bi-filter directives in a BIYACC program can be thought of as specifying 'production priority tables', analogous to the operator precedence tables of, for example, the C programming language (Kernighan and Ritchie, 1989) (chapter *Expressions*) and Haskell (Marlow et al., 2010) (page 51). The main differences (in terms of the parsing direction) are as follows:

- For bi-filters, priority can be assigned independently of associativity and vice versa, while the YACC-style approach does not permit so—by design, when the YACC directives (such as

[1] The YACC-style approach adopts the word *precedence* (Johnson, 1975) while the filter-based approaches tend to use the word *priority* (Klint and Visser, 1994; van den Brand et al., 2002). We follow the traditions and use either word depending on the context.

`%left` and `%right`) are used on multiple tokens, they necessarily specify both the precedence and associativity of those tokens.

- For bi-filters, priority and associativity directives may be used to specify more than one production priority tables, making it possible to put unrelated operators in different tables and avoid (unnecessarily) specifying the relationship between them. It is impossible to do so with the YACC-style approach, for its concise syntax only allows a single operator precedence table.

Our bi-filter directives repair CSTs violating priority and associativity constraints by adding parentheses—for example, if the production of addition expressions in Figure 4.1 is left-associative, then we can repair $^\sharp$`Plus 1 (Plus 2 3)` by adding parentheses around the right subtree, yielding $^\sharp$`Plus 1 (Paren (Plus 2 3))`, provided that the grammar has a production of parentheses annotated with the *bracket attribute* (van den Brand and Visser, 1996; Visser, 1997b):

```
Expr -> ...
      | [Paren]  '(' Expr ')' {# Bracket #} .
```

It instructs our bi-filter directives to use this production when parentheses need to be added. Internally, from the production and bracket attribute annotation, a type class AddParen and corresponding instances for each datatype generated from concrete syntax (Expr for this example) are automatically created:

```
class AddParen t where
  canAddPar :: t -> Bool
  addPar    :: t -> t
```

where `canAddPar` tells whether a CST can be wrapped in a parenthesis structure and `addPar` adds that structure if it is possible or behaves as an identity function otherwise. This makes it possible to automatically generate bi-filters to repair incorrect CSTs (and help the user to define their own bi-filters more easily—see Section 4.5).

In order for bi-filter directives to work correctly, the user should notice the following requirements: (i) Directives shall not mention the parenthesis production annotated with bracket attribute so that they *respect* each other and work properly (as introduced in Definition 4.4.3). (ii) Suppose that the parenthesis production is $NT \longrightarrow \alpha NT_R \beta$ where α and β denote a sequence of terminals—for instance, `Expr -> '(' Expr ')'` above— there shall be exactly one printing action defined for the parenthesis production in the form of $v \mathrel{+\!\!>} \alpha[v \mathrel{+\!\!>} NT_R]\beta$ for the PassThrough

property to hold: for any CST, the (added) parenthesis structure will all be dropped through the conversion to its AST.

Next we introduce our priority and associativity directives and their bi-filter semantics. From a directive, we first generate a bi-filter that checks and repairs only the top of a tree; this bi-filter is then lifted to check and repair all the subtrees in a tree. In the following we will give the semantics of the directives in terms of the generation of the top-level bi-filters, and then discuss the lifted bi-filters and other important properties they satisfy in Section 4.4.3.

Priority Directives

A priority directive defines relative priority between two productions; it removes (in the parsing direction) or repairs (in the printing direction) CSTs in which a node of (relatively) lower priority is a direct child of the node of (relatively) higher priority. For instance, we can define that (the production of) multiplication has higher priority than (the production of) addition for the grammar in Figure 4.1 by writing

```
Expr -> Expr '*' Expr  >  Expr -> Expr '+' Expr ;   or just   Times > Plus ; .
```

The directive first produces the following top-level bi-filter: [1]

```
fTimesPlusPrio  (Times t1 t2 t3) =
  case or [match t1 p, match t2 p, match t3 p, False] of
    False -> Nothing
    True  -> Just (Times (if match t1 p then addPar t1 else t1)
                         (if match t2 p then addPar t2 else t2)
                         (if match t3 p then addPar t3 else t3))
  where p = Plus undefined undefined undefined .
```

We first check whether any of the subtrees `t1`, `t2`, and `t3` violates the priority constraint, i.e. having `Plus` as its top-level constructor—this is checked by the `match` function, which compares the top-level constructors of its two arguments. The resulting boolean values are aggregated using the list version of logical disjunction *or* :: [Bool] → Bool. If there is any incorrect part, we repair it by inserting a parenthesis structure using `addPar`.

[1] Although terminals such as `'*'` and `'+'` are uniquely determined by constructors and not explicitly included in the CSTs, there are fields in CSTs for holding whitespace after them. Thus `Times` still has three subtrees. Also, for simplicity, the bi-filter `fTimesPlusPrio` attempts to repair the whitespace subtree `t2` even though the repair can never happen since `t2` cannot match `p`.

In general, the syntax of priority directives is

$$
\begin{aligned}
\mathit{Priority} &::= \text{`}\texttt{Priority:}\text{'}\ \mathit{PDirective}^{+} \\
\mathit{PDirective} &::= \mathit{ProdOrCons}\ \text{`}\texttt{>}\text{'}\ \mathit{ProdOrCons}\ \text{`}\texttt{;}\text{'} \\
&\quad \mid \mathit{ProdOrCons}\ \text{`}\texttt{<}\text{'}\ \mathit{ProdOrCons}\ \text{`}\texttt{;}\text{'} \\
\mathit{ProdOrCons} &::= \mathit{Prod} \mid \mathit{Constructor} \\
\mathit{Prod} &::= \mathit{Nonterminal}\ \text{`}\texttt{->}\text{'}\ \mathit{Symbol}^{+}
\end{aligned}
$$

where *Constructor* and *Symbol* are already defined in Figure 3.3; for each priority declaration, we can use either *productions* or their names (i.e. constructors).

If the user declares that a production $NT_1 \longrightarrow RHS_1$ has higher priority than another production $NT_2 \longrightarrow RHS_2$, the following priority bi-filter will be generated:

$$
\begin{aligned}
&\textsc{toPrioFilter}[\![(RHS_1, NT_1, RHS_2, NT_2)]\!] = \\
&\quad \text{`}\texttt{f}\text{'}\ conRHS_1\ conRHS_2 \text{`}\texttt{Prio}\text{'}\ \text{`}\texttt{(}\text{'}\ conRHS_1\ \textsc{fillVars}(RHS_1)\text{`}\texttt{)}\ \texttt{=}\text{'} \\
&\qquad \text{`}\texttt{case or [}\text{'}\ \left\langle \text{`}\texttt{match}\text{'}\ t\ \text{`}\texttt{p,}\text{'} \;\middle|\; t \in \textsc{fillVars}(RHS_1) \right\rangle \text{`}\texttt{False}\text{'}\texttt{]}\ \texttt{of}\text{'} \\
&\qquad\quad \text{`}\texttt{False -> Nothing}\text{'} \\
&\qquad\quad \text{`}\texttt{True  -> Just (}\text{'}\ conRHS_1\ \left\langle \textsc{repair}(t) \;\middle|\; t \in \textsc{fillVars}(RHS_1) \right\rangle \text{`}\texttt{)}\text{'} \\
&\qquad \text{`}\texttt{where p = }\text{'}\ \textsc{con}(NT_2, RHS_2)\ \textsc{fillUndefined}(RHS_2) \\
&\quad \text{`}\texttt{f}\text{'}\ conRHS_1\ conRHS_2 \text{`}\texttt{Prio}\text{'}\ \text{`}\texttt{\_}\text{'}\ \text{`}\texttt{=}\text{'}\ \text{`}\texttt{Nothing}\text{'} \\
&\textsc{repair}(t) = \text{`}\texttt{(if match}\text{'}\ t\ \text{`}\texttt{p}\text{'}\ \text{`}\texttt{then addPar}\text{'}\ t\ \text{`}\texttt{else}\text{'}\ t\text{`}\texttt{)}\text{'} \\
&conRHS_1 = \textsc{con}(NT_1, RHS_1) \\
&conRHS_2 = \textsc{con}(NT_2, RHS_2)\ .
\end{aligned}
$$

con looks up constructor names for input productions (divided into nonterminals and right-hand sides); fillVars(nt) generates variable names for each terminal and nonterminal in nt (here RHS_1); fillUndefined is similar to fillVars but it produces `undefined` values instead. If productions are referred to using their constructors, we can simply look up the nonterminals and right-hand sides and use the same code generation strategy.

Transitive Closures. In the same way as conventional Yacc-style approaches, the priority directives are considered transitive. For instance,

```
Expr -> Expr '*' Expr  >  Expr -> Expr '+' Expr ;
Expr -> Expr '+' Expr  >  Expr -> Expr '&' Expr ;
```

implies that `Expr -> Expr '*' Expr > Expr -> Expr '&' Expr ;`. The feature is important in practice since it greatly reduces the amount of routine code the user needs to write for large grammars.

Associativity Directives

Associativity directives assign (left- or right-) associativity to productions. A left-associativity directive bans (or repairs, in the printing direction) CSTs having the pattern in which a parent and its right-most subtree are both left-associative, if the (relative) priority between the parent and the subtree is not defined; a right-associativity directive works symmetrically.

As an example, we can declare that both addition and subtraction are left-associative (for the grammar in Figure 4.1) by writing

```
Left: Expr -> Expr '+' Expr, Expr -> Expr '-' Expr;
```

or just `Left: Plus, Minus;`. Since the relative priority between `Plus` and `Minus` is not defined, we generate top-level bi-filters for all the four possible pairs formed out of `Plus` and `Minus`:

```
fPlusPlusLAssoc (Plus t1 t2 t3) =
  if match t3 p then Just (Plus t1 t2 (addPar t3)) else Nothing
  where p = Plus undefined undefined undefined
fPlusPlusLAssoc _               = Nothing

fMinusMinusLAssoc (Minus t1 t2 t3) =
  if match t3 p then Just (Minus t1 t2 (addPar t3)) else Nothing
  where p = Minus undefined undefined undefined
fMinusMinusLAssoc _                = Nothing


fPlusMinusLAssoc (Plus t1 t2 t3) =
  if match t3 p then Just (Plus t1 t2 (addPar t3)) else Nothing
  where p = Minus undefined undefined undefined
fPlusMinusLAssoc _               = Nothing

fMinusPlusLAssoc (Minus t1 t2 t3) =
  if match t3 p then Just (Minus t1 t2 (addPar t3)) else Nothing
  where p = Plus undefined undefined undefined
fMinusPlusLAssoc _                = Nothing .
```

For instance, `fPlusPlusLAssoc` accepts ♯`Plus (Plus 1 2) 3` but not ♯`Plus 1 (Plus 2 3)`, which is repaired to ♯`Plus 1 (Paren (Plus 2 3))`.

Generally, the syntax of associativity directives is

$$\begin{aligned} \mathit{Associativity} &::= \text{`}\texttt{Associativity:}\text{'}\ \mathit{LeftAssoc}\ \mathit{RightAssoc} \\ \mathit{LeftAssoc} &::= \text{`}\texttt{Left:}\text{'}\ \mathit{ProdOrCons}^{+}\{\text{`}\texttt{,}\text{'}\}\ \text{`}\texttt{;}\text{'} \\ \mathit{RightAssoc} &::= \text{`}\texttt{Right:}\text{'}\ \mathit{ProdOrCons}^{+}\{\text{`}\texttt{,}\text{'}\}\ \text{`}\texttt{;}\text{'}\ . \end{aligned}$$

Now we explain the generation of (top-level) bi-filters from associativity directives. We will consider only left-associativity directives, as right-associativity directives are symmetric. For every pair of left-associative productions whose relative priority is not defined—including cases where the two productions are the same—we generate a bi-filter to repair CSTs whose top uses the first production and whose right-most child uses the second production. Let $NT_1 \longrightarrow \alpha_1 NT_{1R}$ and $NT_2 \longrightarrow \alpha_2 NT_{2R}$ be two such productions, where α_1 (α_2) matches a sequence of arbitrary symbols of any length and NT_{1R} (NT_{2R}) is the right-most symbol and must be a nonterminal. (If it is not a nonterminal, it is meaningless to discuss associativity.) The generated bi-filter is

$$\begin{aligned} &\textsc{toLAssocFilter}[\![\alpha_1 NT_{1R}, NT_1, \alpha_2 NT_{2R}, NT_2]\!] = \\ &\quad \text{`}\texttt{f}\text{'}\ \mathit{conRHS}_1\ \mathit{conRHS}_2\ \text{`}\texttt{LAssoc}\text{'}\ \text{`}\texttt{(}\text{'}\mathit{conRHS}_1\ \textsc{fillVars}(\alpha_1 NT_{1R})\ \text{`}\texttt{)}\text{'}\ \text{`}\texttt{=}\text{'} \\ &\qquad \text{`}\texttt{if match }\text{'}\ \mathit{ntrVar}\ \text{`}\texttt{ p}\text{'} \\ &\qquad\quad \text{`}\texttt{then Just (}\text{'}\ \mathit{conRHS}_1\ \textsc{fillVars}(\alpha_1)\ \text{`}\texttt{(addPar}\text{'}\ \mathit{ntrVar}\ \text{`}\texttt{))}\text{'} \\ &\qquad\quad \text{`}\texttt{else Nothing}\text{'} \\ &\qquad \text{`}\texttt{where p = }\text{'}\mathit{conRHS}_2\ \textsc{fillUndefined}(\alpha_2 NT_{2R}) \\ &\quad \text{`}\texttt{f}\text{'}\ \mathit{conRHS}_1\ \text{`}\texttt{LAssoc}\text{'}\ \text{`}\texttt{\_}\text{'}\ \text{`}\texttt{=}\text{'}\ \text{`}\texttt{Nothing}\text{'} \\ &\mathit{conRHS}_1 = \textsc{con}(NT_1, \alpha_1 NT_{1R}) \\ &\mathit{conRHS}_2 = \textsc{con}(NT_2, \alpha_2 NT_{2R}) \\ &\mathit{ntrVar} = \textsc{fillVarsFrom}(\textsc{length}(\alpha_1), NT_{1R})\ . \end{aligned}$$

Functions con, fillUndefined, and fillVar have the same behaviour as before; fillVarsFrom (which is a variation of fillVars) generates variable names for each terminal and nonterminal in its argument with suffix integers counting from a given number to avoid name clashing.

Handling Injective Productions. Sometimes the grammar may contain injective productions (also called chain productions) (van den Brand et al., 2002), which have only a single nonterminal

on their right-hand side, like `InfE -> [FromE] Exp`. When we use it to define a grammar

```
InfE -> [FromE] Exp
Exp  -> [Plus]  InfE '+' InfE
      | [Times] InfE '*' InfE ,
```

program text `1 + 2 * 3` will be parsed to two CSTs, namely $\texttt{cst}_1$ = `♯Plus (FromE 1) (FromE (Times 2 3))` and $\texttt{cst}_2$ = `♯Times (FromE (Plus 1 2) (FromE 3))`, and we want to spot $\texttt{cst}_2$ and discard it using the priority directive `Times > Plus`. If handled naively, the bi-filter generated from the directive would only remove CSTs having pattern `Times (Plus _ _) _` (and two other similar ones), but $\texttt{cst}_2$ would not match the pattern due to the presence of the `FromE` node between `Times` and `Plus`. We made some effort in the implementation to make the `match` function ignore the nodes corresponding to injective productions (`FromE` in this case).

4.4.3 Properties of the Generated Bi-Filters

We discuss some properties of the bi-filters generated from our priority and associativity directives, to justify that it is safe to use these bi-filters without disrupting the well-behavedness of the whole system. Specifically:

- The generated top-level bi-filters satisfy RepairJudge, and it is easy to write actions to make them satisfy PassThrough.
- The bi-filters lifted from the top-level bi-filters still satisfy RepairJudge and PassThrough.
- The lifted bi-filters are *commutative*, which not only implies that all such bi-filters respect each other and can be composed in any order, but also guarantees that we do not have to worry about the order of composition since it does not affect the behaviour.

We will give only high-level, even informal, arguments for these properties, since, due to the generic nature of the definitions of these bi-filters (in terms of *Scrap Your Boilerplate* (Lämmel and Jones, 2003)), to give formal proofs we would have to introduce rather complex machinery (e.g., datatype-generic induction), which would be tedious and distracting.

Top-level bi-filters. The fact that the generated top-level bi-filters satisfy RepairJudge can be derived from the requirement that the directives do not mention the parenthesis production. Because of the requirement, in the generated bi-filters, repairing is always triggered by `match`ing a non-parenthesis production, and after that repairing will not be triggered again because a parenthesis production will have been added. For example, in the bi-filter `fTimesPlusPrio` (in

Section 4.4.2), with `match t1 p`, `match t2 p`, and `match t3 p` we check whether `t1`, `t2`, and `t3` has `Plus` as the top-level production, which is different from the parenthesis production `Paren`; if any of the `match`ing succeeds, say `t1`, then `addPar t1` will add `Paren` at the top of `t1`, and `match (addPar t1) p` is guaranteed to be `False`, so the subsequent invocation of `judge fTimesPlusPrio` will return `True`. For PassThrough, since all the top-level bi-filters do is add parenthesis productions, we can simply make sure that appearances of the parenthesis production are ignored by *get*, i.e. *get* (*addPar s*) = *get s* for all *s*; this, by well-behavedness, is the same as making *put* (printing actions) skip over parentheses. For example, for the grammar in Figure 4.1, we should write `t +> '(' [t +> Expr] ')'` as the only printing action mentioning parentheses, which means that *put* (Paren *s*, *t*) = *fmap* Paren (*put* (*s*, *t*)) for all *s* and *t*. Then the following reasoning implies that *get* (Paren *s*) = *get s* for all *s*:

$$
\begin{aligned}
& \mathit{get}\ (\mathsf{Paren}\ s) = \mathsf{Just}\ t \\
\Leftrightarrow\quad & \{\ \Rightarrow \text{by Hippocraticness and} \Leftarrow \text{by Correctness}\ \} \\
& \mathit{put}\ (\mathsf{Paren}\ s,\ t) = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow\quad & \{\ \text{By the above statement: } \mathit{put}\ (\mathsf{Paren}\ s,\ t) = \mathit{fmap}\ \mathsf{Paren}\ (\mathit{put}\ (s,\ t))\ \} \\
& \mathit{fmap}\ \mathsf{Paren}\ (\mathit{put}\ (s,\ t)) = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow\quad & \{\ \text{Lemma 4.3.6 and the definition of } \mathit{fmap}\ \} \\
& \mathit{put}\ (s,\ t) = \mathsf{Just}\ s \\
\Leftrightarrow\quad & \{\ \Rightarrow \text{by Correctness and} \Leftarrow \text{by Hippocraticness}\ \} \\
& \mathit{get}\ s = \mathsf{Just}\ t
\end{aligned}
$$

for all *s* and *t*.

Lifted bi-filters. The lifted bi-filters apply the top-level bi-filters to all the subtrees in a CST in a bottom-up order. Formally, we can define, datatype-generically, a lifted bi-filter as a composition of top-level bi-filters, and use datatype-generic induction to prove that there is suitable respect among the top-level bi-filters being composed, and that the lifted bi-filter satisfies RepairJudge and PassThrough if the top-level ones do. But here we provide only an intuitive argument. What the lifted bi-filters do is find all prohibited pairs of adjoining productions and separate all the pairs by adding parenthesis productions. For RepairJudge, since all prohibited pairs are eliminated after repairing, there will be nothing left to be repaired in the resulting CST, which will therefore be deemed valid. For PassThrough, the intuition is the same as that for the top-level bi-filters.

Commutativity. Composite bi-filters $i \triangleleft j$ and $j \triangleleft i$ may have different behaviour, so in general we need to know the order of composition to figure out the exact behaviour of a composite bi-filter. This can be difficult when using our bi-filter directives, since a lot of bi-filters are implicitly generated from the directives, and it is not straightforward to specify the order in which all the explicitly and implicitly generated bi-filters are composed. Fortunately we do not need to do so, for all the bi-filters generated from the directives are *commutative*, meaning that the order of composition does not affect the behaviour.

Definition 4.4.5 (Bi-Filter Commutativity)**.** Two bi-filters i and j are *commutative* exactly when

$$repair_i \circ repair_j = repair_j \circ repair_i \,.$$

By Lemma 4.4.2, this implies $repair\,(i \triangleleft j) = repair\,(j \triangleleft i)$. Note that $judge\,(i \triangleleft j) = judge\,(j \triangleleft i)$ by definition, so we do not need to require this in the definition of commutativity.

An important fact is that commutativity is stronger than respect, so it is always safe to compose commutative bi-filters.

Lemma 4.4.6. Commutative bi-filters respect each other.

Proof. Given commutative bi-filters i and j, we show that j respects i. Suppose that $judge_i\ t = \mathsf{True}$ for a given tree t. Then

$$\begin{aligned}
& judge_i\ (repair_j\ t) \\
= \quad & \{\ repair_i\ t = t, \text{since } judge_i\ t = \mathsf{True}\ \} \\
& judge_i\ (repair_j\ (repair_i\ t)) \\
= \quad & \{\ i \text{ and } j \text{ are commutative}\ \} \\
& judge_i\ (repair_i\ (repair_j\ t)) \\
= \quad & \{\ \text{RepairJudge}\ \} \\
& \mathsf{True}\,.
\end{aligned}$$

It follows by symmetry that i respects j as well. □

Now let us consider why any two different lifted bi-filters are commutative. (Commutativity is immediate if the two bi-filters are the same.) There are two key facts that lead to commutativity: (i) repairing does not introduce more prohibited pairs of productions, and (ii) the prohibited pairs of adjoining productions checked and repaired by the two bi-filters are necessarily different.

Therefore the two bi-filters always repair different parts of a tree, and can repair the tree in any order without changing the final result. Fact (i) is, again, due to the requirement that the directives do not mention the parenthesis production, which is the only thing we add to a tree when repairing it. Fact (ii) can be verified by a careful case analysis. For example, we might be worried about the situation where a left-associative directive looks for production Q used at the right-most position under production P, while a priority directive also similarly looks for Q used under P, but the two directives cannot coexist in the first place since the first directive implies P and Q have no relative priority whereas the second one implies Q has lower priority than P.

4.5 Manually Written Bi-Filters

There are some other ambiguities that our directives cannot eliminate. In these cases, the user can define their own bi-filters and put them in the `#OtherFilters` part in a BiYacc program as shown in Figure 3.3. The syntax is

OtherFilters ::= '`[`' *HsFunDecl*$^+${'`,`'} '`]`' *HsCode*
HsFunDecl ::= *HsFunName* ' `:: BiFilter` ' *Nonterminal* .

That is, this part of the program begins with a list of declarations of the names and types of the user-defined bi-filters, whose Haskell definitions are then given below.

Now we demonstrate how to manually write a bi-filter by resolving the ambiguity brought by the dangling else problem. But before that, let us briefly review the problem, which arises, for example, in the following grammar:

```
Exp ->  [ITE] 'if' Exp 'then' Exp 'else' Exp
     |  [IT]  'if' Exp 'then' Exp .
```

With respect to this grammar, the program text `if a then if x then y else z` can be recognised as either `if a then (if x then y else z)` or `if a then (if x then y) else z`. To resolve the ambiguity, usually we prefer the 'nearest match' strategy (which is adopted by Pascal, C, and Java): `else` should match its nearest `then`, so that `if a then (if x then y else z)` is the only correct interpretation.

The user may think that the problem can be solved by a priority (bi-)filter `ITE > IT;`, in the hope that the production *if-then-else* binds tighter than the production *if-then*. Unfortunately, this is incorrect as pointed out by Klint and Visser (1994), because the corresponding (bi-)filter

incorrectly rules out the pattern $^\sharp$`ITE _ _ (IT _ _)`, which prints to unambiguous text, e.g., `if a then b else if x then y`. In fact, the (dangling else) problem is tougher than one might think and cannot be solved by any (bi-)filter performing pattern matching with a fixed depth (Klint and Visser, 1994).

Klint and Visser proposed an idea to disambiguate the dangling-else grammar: Let Greek letters $\alpha, \beta, \ldots$ match a sequence of symbols of any length. Then the program text `if` α `then` β `else` γ should be banned if the right spine of β contains any `if` ψ `then` ω, as shown in the paper (Klint and Visser, 1994). With the full power of (bi-)filters, which are fully-fledged Haskell functions, we can implement this solution in the following bi-filter:

```
fCond (ITE c1 e1 e2) = case checkRightSpine e1 of
  True  -> Nothing
  False -> Just (ITE c1 (addPar e1) e2)

-- collect the names of the constructors in the right spine and
-- check if the collected constructors contain "IT"
checkRightSpine t = ... .
```

This bi-filter is commutative with the bi-filters generated from our directives, for it (i) only searches for non-parenthesis productions that are not declared in any other directives, and (ii) inserts only a parenthesis production when repairing incorrect CSTs. The reader may find the code of `checkRightSpine` in more detail in Figure 4.5. One remark is that wrapping the whole `e1` into a parenthesis production is a lazy man's way, which is correct but does not insert parentheses to the most proper place—for instance, to repair `if x then a + if b then c else d`, both `if x then ( a + if b then c) else d` and `if x then a + (if b then c) else d` are correct, and the former is our lazy man's way.

4.6 Case Study Using Ambiguous Tiger

In Section 3.3, we have shown how to obtain a consistent pair of parser and retentive printer of Tiger in unambiguous grammars using BiYacc, and demonstrated the power of retentive printers by examples about syntactic sugar, resugaring, and language evolution. Now, we (re-)define the Tiger language using ambiguous grammars and bi-filters. The examples about syntactic sugar, resugaring, and language evolution basically remain the same, so we mainly ignore them and only point out the difference.

```
#Concrete
SeqExp  -> '{'        '}' | '{' ExpSeq '}' ;
ExpSeq  -> Exp ';' ExpSeq | Exp ;

Prmtv  -> [Paren] '(' Exp ')' {# Bracket #} | CallExp | SeqExp | ...
        | [Or]    Prmtv '|' Prmtv  |  [And]   Prmtv '&' Prmtv
        | [Plus]  Prmtv '+' Prmtv  |  [Times] Prmtv '*' Prmtv | ...
        | [Neg]   '-' Prmtv | Numeric  | String | LValue | 'nil' ;

IfThenElse -> [ITE] 'if' Exp 'then' Exp 'else' Exp ;
IfThen     -> [IT]  'if' Exp 'then' Exp ;
...
```

Figure 4.4: An excerpt of Tiger's ambiguous concrete syntax.

Excerpts of the concrete syntax of ambiguous Tiger are shown in Figure 4.4. Here, we add a parenthesis production to the grammar (and discard it when converting CSTs to ASTs, so that the PassThrough property could be satisfied), for Tiger's original grammar has no parenthesis production and an expression within round parentheses is regarded as a singleton expression sequence. This modification also makes it necessary to change the enclosing brackets for expression sequences from round brackets `()` to curly brackets `{}`, which helps (LALR(1) parsers) to distinguish a singleton expression sequence from an expression within parentheses.

Following Hirzel and Rose (2013)'s specification, the disambiguation directives for Tiger are shown in Figure 4.5; for instance, we define multiplication to be left-associative. The directives also include a concrete treatment of the dangling else problem, which is usually 'not solved' when using a Yacc-like (LA)LR parser generator to implement parsers: in this case, rather than resolving the grammatical ambiguity, we often rely on the default behaviour of the parser generator—preferring shift.

Modifications to the Syntactic Sugar Example. In accordance with the changes to the Tiger's concrete syntax, the printing actions for this example now becomes

```
TExp +> Prmtv
  TCond e1 (TInt 1) (JJ e2)  +>  [e1 +> Prmtv] '|' [e2 +> Prmtv];
  TCond e1 e2 (JJ (TInt 0))  +>  [e1 +> Prmtv] '&' [e2 +> Prmtv]; .
```

```
#Directives
Priority:
Times >  Plus ;
And   >  Or   ;
...

Associativity:
Left:  Times, Plus, And  ... ;
Right: Assign,  ... ;

#OtherFilters
[ fDanglingElse :: BiFilter IfThenElse ]

fDanglingElse (ITE t1 exp1 t2 exp2 t3 exp3) =
  case checkRightSpine exp2 of
    True  -> Nothing
    False -> Just (ITE t1 exp1 t2 (addPar exp2) t3 exp3)

checkRightSpine t = let spineStrs = getRSpineCons t
                    in  and $ map (\str -> str /= "IT") spineStrs

class GetRSpineCons t where
  getRSpineCons :: t -> [String]

instance GetRSpineCons IfThenElse where
  getRSpineCons (ITE _ _ _ _ _ r) = ["ITE"] ++ getRSpineCons r

instance GetRSpineCons IfThen where
  getRSpineCons (IT _ _ _ r) = ["IT"] ++ getRSpineCons r

instance GetRSpineCons LetExp where
  getRSpineCons (LetExp1 _ _ _ _ _) = ["LetExp1"]

...
```

Figure 4.5: An excerpt of the disambiguation directives for Tiger. (A type class `GetRSpineCons` is defined and implemented for collecting the constructors on the right spine of a given tree. Function `getRSpineCons` is recursively invoked for CSTs whose right-most subtree is (parsed from) a nonterminal.)

4.7 Related Work

The grammar of a programming language is usually designed to be unambiguous. Various parser-dependent disambiguation methods such as grammar transformation (LaLonde and des Rivieres, 1981) and parse table conflicts elimination (Johnson, 1975) have been developed to guide the parser to produce a single correct CST (Klint and Visser, 1994). On the other hand, natural languages that are inherently ambiguous usually require their parsing algorithms to produce all the possible CSTs; this requirement gives rise to algorithms such as Earley (Earley, 1970) and generalised LR (Tomita, 1985) (GLR for short). Although these parsing algorithms produce all the possible CSTs, both their time complexity and space complexity are reasonable. For instance, GLR runs in cubic time in the worst situation and in linear time if the grammar is 'almost unambiguous' (Scott et al., 2007).

The idea to relate generalised parsing with parser-independent disambiguation for programming languages is proposed by Klint and Visser (1994). According to Klint and Visser, filters can be classified as lexical level ones and context-free level ones. As their names suggest, lexical filters are used in the tokenising phase to prefer longest match and identifying keywords while context-free filters are used to remove unwanted parses in the parsing phase[1]. For context-free level filters, they are further classified as two kinds: *property filters* (defined in terms of predicates on a single tree) and *comparison filters* (defined in terms of relations among trees), but we only adapted and bidirectionalised predicate filters in this thesis. One difficulty lies in the fact that it is unclear how to define the *repair* function for comparison filters, as they generally select better trees rather than absolutely correct ones—in the printing direction, since *put* only produces a single CST, we do not know whether this CST needs repairing or not (for there is no other CST to compare). This is also one of the most important problems for our future work.

Parser-independent disambiguation (for handling priority and associativity conflicts) can also be found in LaLonde and des Rivieres's (LaLonde and des Rivieres, 1981) and Aasa's (Aasa, 1995) work. At first glance, our *repair* function is quite similar to LaLonde and des Rivieres's post-parse *tree transformations* that bring a CST into an *expression tree*, on whose nodes additional restrictions of priority and associativity are imposed. To be simple (but not completely precise), a CST's corresponding expression tree is obtained by first dropping all the nodes constructed from injective productions[2] (note that parentheses nodes are still kept) and then use a *precedence-introducing tree transformation* to reshape the result. The transformation will do 'repairing' by

[1] Although using advanced techniques, tokenising and parsing can be merge into a single phase to become scannerless parsing (van den Brand et al., 2002), we treat them as two separate phases in this thesis.

[2] An injective production, or a chain production, is one whose right-hand side is a single nonterminal; for instance, `E -> N`.

rotating all the adjacent nodes of the tree where priority or associativity constraint is violated. By contrast, our *repair* function is simpler and only introduces parentheses in places where the *judge* function returns False. In short, their tree transformations are a kind of parser-independent disambiguation which does not require generalised parsing; however, those tree transformations are (almost) not applicable in the printing direction if well-behavedness is taken into consideration (due to the rotation of CSTs). But there is no need for LaLonde and des Rivieres to consider the printing direction, either. Furthermore, it is not clear whether their approach can be generalised to handle other types of conflicts rather than the ones caused by priority and associativity.

There is much research on how to handle ambiguity in the parsing direction as discussed above; conversely, little research is conducted for 'handling ambiguity in the printing direction' and we find only one paper (van den Brand and Visser, 1996) that describes how to produce correct program text regarding priority and associativity, which is also one of the bases of our work. We extend their work (van den Brand and Visser, 1996) by allowing the bracket attribute to work with injective productions such as `E -> T; T -> F; F -> '(' E ')' {# Bracket #};`. (The previous work seems to only support the bracket attribute in the form of `E -> '(' E ')' {# Bracket #};`; whether the nonterminal `E` on the left-hand side and right-hand side can be different is not made clear.)

Here we also briefly discuss ambiguity detection for the filter approaches: Priority and associativity (bi-)filters can be applied to (LA)LR parse tables to resolve (shift/reduce) conflicts (Klint and Visser, 1994; van den Brand et al., 2002; Visser, 1997a,b), and thus the completeness for simple (bi-)filters (see Definition 4.3.2) on LALR(1) grammars can be statically checked. However, our implementation does not support it, for bi-filter directives are more general, as stated in the beginning of Section 4.4.2, and therefore cannot be transformed to the underlying parser generator's Yacc-style directives. Finding a way to directly apply priority and associativity bi-filters to parse tables (generated by Happy) is left as future work.

Finally, we compare our approach with the conventional ones in general. In history, a printer is believed to be much simpler than a parser and is usually developed independently (of its corresponding parser). While a few printers choose to produce parentheses at every occasion naively, most of them take disambiguation information (for example, from the language's operator precedence table) into account and try to produce necessary parentheses only. However, as the Yacc-style conventional disambiguation (Johnson, 1975) is parser-dependent, this parentheses-adding technique is also printer-dependent. As the post-parse disambiguation increases the modularity of the (front end of the) compiler (LaLonde and des Rivieres, 1981), we believe that our post-print parentheses-adding increases the modularity once again. Additionally, the unification

of disambiguation for both parsing and printing makes it possible for us to impose bi-filter laws, which further makes it possible to guarantee the well-behavedness of the whole system.

4.8 Discussions

In this chapter, we made an extension to our solution (mainly) for the isomorphism part. We proposed bi-filters and extended BiYacc by incorporating bi-filters while still respecting the consistency of the generated parser and printer pairs. We provided compositional bi-filter directives which compile to priority and associativity bi-filters and demonstrated how to manually write bi-filters. Here, we briefly discuss several issues: bi-filters as asymmetric lenses, efficient implementation of bi-filters, and relaxation of the Correctness law.

Bi-Filters as Asymmetric Lenses

Although it is not necessary, bi-filters can be regarded as a special kind of asymmetric lenses between CSTs and boolean values. Given a bi-filter F, we use get_F and put_F to denote the *get* and *put* functions (partially) applied to F and give their semantics:

$$
\begin{aligned}
&get_F :: \mathsf{CST} \to \mathsf{Maybe\ Bool} \\
&get_F\ s = \mathsf{Just}\ (judge_F\ s) \\
&put_F :: \mathsf{CST} \to \mathsf{Bool} \to \mathsf{Maybe\ CST} \\
&put_F\ s\ \mathsf{False} \mid get\ s == \mathsf{Just\ True} = \mathsf{Just}\ defVal \\
&put_F\ s\ \mathsf{False} \mid get\ s == \mathsf{Just\ False} = \mathsf{Just}\ s \\
&put_F\ s\ \mathsf{True} = \mathsf{Just}\ (repair_F\ s)
\end{aligned}
$$

where *defVal* is a value satisfying $judge_F\ defVal = \mathsf{False}$. (Since many bi-filters are based on pattern matching, it is very easy to create a *defVal* having invalid patterns.)

It is easy to prove the well-behavedness between get_F and put_F with the help of bi-filter laws.

Hippocraticness. Case 1: $get_F\ s = \mathsf{Just\ True}$. Then $put_F\ s\ \mathsf{True} = \mathsf{Just}\ (repair_F\ s) = \mathsf{Just}\ s$ by JudgeRepair.

Case 2: $get_F\ s = \mathsf{Just\ False}$. Then $put_F\ s\ \mathsf{False} = \mathsf{Just}\ s$.

□

Correctness. Case 1: The view is False and $get_F\ s$ = Just True. Then Just s' = $put_F\ s$ False = Just *defVal*; $get_F\ s'$ = get_F *defVal* = Just False.

Case 2: The view is False and $get_F\ s$ = Just False. Then Just s' = $put_F\ s$ False = Just s; $get_F\ s'$ = $get_F\ s$ = Just False.

Case 3: The view is True. Then Just s' = $put_F\ s$ True = Just ($repair_F\ s$); $get_F\ s'$ = $get_F\ (repair_F\ s)$ = Just True by RepairJudge.

□

Efficient Bi-Filters

For efficiency concerns, (bi-)filters should be applied as early as possible (Klint and Visser, 1994). As an illustration, if the disambiguation involving associativity and priority of binary operators is delayed until generating all the CSTs, there will be millions of trees awaiting disambiguation for a simple (but highly ambiguous) expression such as `1 + 2 + ··· + 15`. To speed up parsing, we can 'write' the *judge* function of priority and associativity bi-filters into parse tables so that incorrect CSTs in terms of priority and associativity will not be produced at all. The result is the same as applying them to CSTs if injective productions (chains) are automatically handled, as discussed in papers (Klint and Visser, 1994; van den Brand et al., 2002; Visser, 1997a,b) and can be seen in some real-world application (van den Brand et al., 2001).

We showed how to compile disambiguation directives to bi-filters by directly performing pattern-matching on input trees, which brings intuitive semantics makes the correctness easy to verify. However, this approach has a flaw that the generated code is inefficient and, in the worst case, grows quadratically in the number of priority directives. For instance, suppose that (productions) A and B have higher priority than C and D, which further have higher priority than E, F, and G; since priority is transitive, A and B also have higher priority than E, F, and G. These in total contribute to $2 \times 2 + 2 \times 3 + 2 \times 3 = 16$ cases. A better approach is to sort, for each priority table, the priority directives in order and assign to each production an integer representing its priority, so that we can use a generic function comparing the priorities of a parent node and each of its child node.

Relaxation of Correctness and Quotient Lenses

Well-behavedness enforces that (i) if we parse a piece of program text to a tree and print the tree back, we get the same text, and (ii) if we print a tree to program text and parse it back, we get an

identical tree. Statement (i) is perfect (for program text), as we focus on syntactic components of the program text. However, statement (ii) is a little too restrictive for ASTs, for we care more about the semantics of an AST rather than its structure. In other words, different ASTs may have the same semantics and therefore there is no need to force an AST to print and parse to the same tree—as long as the semantics does not change. A possible fix is to define equivalence classes for ASTs according to their semantics and revise the Correctness law to work on equivalence classes. The fix will also significantly advance the use of bi-filters, as the revised PassThrough law working on equivalence classes will permit a large number of *repair* functions that are forbidden before. For instance, in order to comply with PassThrough law, we had adapted TIGER's grammar to let parenthesis structures of a CST be dropped when they are transformed to ASTs. Given that parenthesis structures in ASTs do not affect the semantics and ASTs with and without parenthesis structure belong to the same equivalence class, there is no need to adapt the grammar. The research on lenses for equivalent classes can be found in research papers about quotient lenses (Foster et al., 2008) and BOOMERANG (Bohannon et al., 2008).

5 Retentive Printing

We have introduced our DSL (Chapter 3) and an extension (Chapter 4) that deals with grammatical ambiguity (mainly) for the isomorphism part. In this chapter, we explore another extension to our solution for the lens part.

Over the years, we find that while lenses are designed to retain information—for instance, information such as syntactic sugar and comments in the updated program text—the two well-behavedness laws are not strong enough for guaranteeing information retention in the *put* direction when the view is not consistent with the source (i.e. $get\ s \neq v$). This results in a problem that, in terms of information retention, 'well-designed' lenses lead to good performance while 'bad-designed' ones lead to bad performance, despite the fact that they are all well-behaved. Our DSL BiYacc has the same problem: it is the design of BiYacc which happens to propagate changes properly into program text while keeping comments; it succeeds in some cases but will fail in many others.

Let us first illustrate how well-behaved lenses may behave badly by a very simple example, in which *get* is a projection (function) extracting the first element from a tuple of integers and strings. (Hence a source and a view are consistent if the first element of the source tuple is equal to the view.)

```
get :: (Int, String) -> Int
get (i, s) = i
```

Given this specific `get`, we can define $\texttt{put}_1$ and $\texttt{put}_2$, both of which are well-behaved with `get` but the behaviour is rather different: $\texttt{put}_1$ simply replaces the integer of the source (tuple) with the view, while $\texttt{put}_2$ superfluously sets the string (of the tuple) empty, in silence, when the source (tuple) is not consistent with the view.

```
put₁ :: (Int, String) -> Int -> (Int, String)
put₁ (i, s) i' = (i', s)

put₂ :: (Int, String) -> Int -> (Int, String)
put₂ src    i'  | get src == i'  = src
put₂ (i, s) i'  | otherwise      = (i',  "")
```

From another perspective, $\texttt{put}_1$ retains the string from the old source when performing the update, while $\texttt{put}_2$ chooses to discard that string—which is not desired but 'perfectly legal', for the string does not contribute to the consistency relation. In fact, unexpected behaviour of this kind of well-behaved lenses could even lead to disaster in practice. For instance, relational databases can be thought of as tables consisting of rows of tuples, and well-behaved lenses used for maintaining a database and its view may erase important data after an update, as long as the data does not contribute to the consistency relation (in most cases this is because the data is simply not in the view). This fact seems fatal, as asymmetric lenses have been considered a satisfactory solution to the longstanding view update problem (stated at the beginning of Foster et al.'s seminal paper (Foster et al., 2007)).

The root cause of the information loss (after an update) is that while lenses are designed to retain information, well-behavedness actually says very little about the retention of information: the only law guaranteeing information retention is Hippocraticness, which merely requires that the *whole* source should be unchanged if the *whole* view is. In other words, if we have a very small change on the view, we are free to create any source we like. This is too 'global' in most cases, and it is desirable to have a law that makes such a guarantee more 'locally'.

To have a finer-grained law, we propose *retentive lenses*, an extension of the original lenses, which can guarantee that if parts of the view are unchanged, then the corresponding parts of the source are retained as well. Compared with the original lenses, the *get* function of a retentive lens is enriched to compute not only the view of the input source but also a set of *links* relating corresponding parts of the source and the view. If the view is modified, we may also update the set

of links to keep track of the correspondence that still exists between the original source and the modified view. The *put* function of the retentive lens is also enriched to take the links between the original source and the modified view as input, and it satisfies a new law, *Retentiveness*, which guarantees that those parts in the original source having correspondence links to some parts of the modified view are retained at the right places in the updated source.

In this chapter, we formalise the idea of links and develop a formal definition of retentive lenses (Section 5.1) for tree-based data[1]. To show that retentive lenses are feasible, we present a DSL (Section 5.2) for writing tree synchronisers; we present its syntax, semantics, and also prove that any program written in our DSL gives rise to a pair of retentive *get* and *put*. With the help of several view editing operations that also update the links between the view and the original source (Section 5.3), we demonstrate the usefulness of retentive lenses in practice by presenting case studies on code refactoring, resugaring, and XML synchronisation (Section 5.4). We discuss related work (Section 5.5) regarding various alignment strategies for lenses, provenance and origin between two pieces of data, and operation-based BX. Finally, we briefly discuss (Section 5.6) the (possibility of) integration of Retentiveness into BiYacc, our choices of opting for triangular diagrams and Strong Retentiveness (that subsumes Hippocraticness), our thought on (retentive) lens composition, the feasibility of retaining code styles for refactoring tools, and our choice of the word retentive.

5.1 Retentive Lenses for Trees

In this section, we will start by introducing a 'region model' for decomposing trees (Section 5.1.1), and providing a high-level sketch of what retentive lenses should do (Section 5.1.2). After that, we develop a formal definition of links (Section 5.1.3) and retentive lenses (Section 5.1.4) for algebraic data types (which we call 'trees' or 'terms'), through revising classic lenses (Foster et al., 2007) by

- extending *get* and *put* to incorporate links—specifically, we will make *get* return a collection of consistency links, and make *put* additionally take a collection of input links—and
- adding a finer-grained law Retentiveness, which formalises the statement 'if parts of the view are unchanged then the corresponding parts of the source should be retained'.

To be concrete, we will use the synchronisation of concrete and abstract representations of arithmetic expressions as the running example throughout.

[1] Although we focus on the synchronisation between tree-based data, the structure shall be considered more general than tables in relational databases. (A table is a list of tuples, and both lists and tuples can be encoded as trees.)

```
data Expr   =  Plus   Annot Expr Term   getE :: Expr -> Arith
            |  Minus  Annot Expr Term   getE (Plus   _ e  t) = Add (getE e) (getT t)
            |  FromT  Annot Term        getE (Minus  _ e  t) = Sub (getE e) (getT t)
                                        getE (FromT  _    t) = getT t
data Term   =  Lit    Annot Int
            |  Neg    Annot Term        getT :: Term -> Arith
            |  Paren  Annot Expr        getT (Lit     _ i  ) = Num i
                                        getT (Neg     _ t  ) = Sub (Num 0) (getT t)
type Annot  =  String                   getT (Paren   _ e  ) = getE e

data Arith  =  Add Arith Arith
            |  Sub Arith Arith
            |  Num Int
```

Figure 5.1: Data types for concrete and abstract syntax of the arithmetic expressions and the consistency relations between them as `getE` and `getT` functions in Haskell.

5.1.1 Regions of Trees

Let us first get familiar with the concrete and abstract representations of the running example given in Figure 5.1, a (simplified) variation of the arithmetic expressions in Figure 3.1. Here, data types of CSTs are explicitly defined and all the constructors have an annotation field of type `Annot` for holding comments. The two concrete types `Expr` and `Term` coalesce into the abstract representation type `Arith`, which does not include annotations, explicit parentheses, and negations. The consistency relations between CSTs and ASTs are defined in terms of the *get* functions—`e :: Expr` (resp. `t :: Term`) is consistent with `a :: Arith` exactly when `getE e = a` (resp. `getT t = a`).

As mentioned in the introduction, the core idea of Retentiveness is to use links to relate parts of the source and view. For trees, a straightforward interpretation of a 'part' is a subtree of the data. But it is too restrictive in most cases, and a more useful interpretation of a 'part' is a *region* of a tree, i.e. a partial subtree. Partial trees are trees where some subtrees can be missing. We will describe the content of a partial tree with a pattern that contains wildcards at the positions of missing subtrees. In Figure 5.2, all grey areas are examples of regions; the topmost region in `cst` is located at the root of the whole tree, and its content has the pattern `Plus "a plus" _ _`, which says that the region includes the `Plus` node and the annotation `"a plus"`, but not the other two subtrees with roots `Minus` and `Neg` matched by the wildcards.

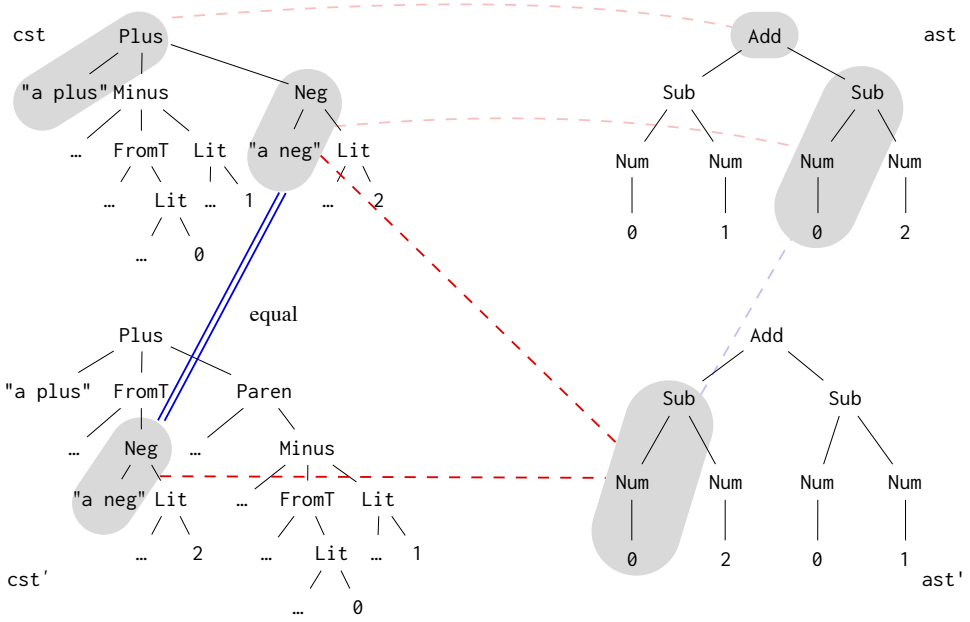


Figure 5.2: The triangular guarantee. (Grey areas are some of the possible regions. Two light dashed lines between `cst` and `ast` are two of the consistency links produced by `getE cst`. When updating `cst` with `ast'`, the region `Neg "a neg" _` connected by the red dashed diagonal link is guaranteed to be preserved in the result `cst'`, and `getE cst'` will link the preserved region to the same region `Sub (Num 0) _` of `ast'`.)

5.1.2 A Sketch of Retentive Lenses

Having broken up source and view trees into regions, we can put in *links* to record the correspondences between source and view regions. In Figure 5.2, for example, the light red dashed lines between the source `cst` and the view `ast = getE cst` represent two possible links. The topmost region of pattern `Plus "a plus" _ _` in `cst` corresponds to the topmost region of pattern `Add _ _` in `ast`, and the region of pattern `Neg "a neg" _` in the right subtree of `cst` corresponds to the region of pattern `Sub (Num 0) _` in `ast`. The *get* function of a retentive lens will be responsible for producing an initial set of links between a source and its view.

As the view is modified, the links between the source and view should also be modified to reflect the latest correspondences between regions. For example, in Figure 5.2, if we change `ast` to `ast'` by swapping the two subtrees under `Add`, then there should be a new link (among others) recording the fact that the `Neg "a neg" _` region and the `Sub (Num 0) _` region are still related. In general, the collection of diagonal links between `cst` and `ast'` may be created in many ways, such as directly comparing `cst` and `ast'` and producing links between matching areas, or composing

the light red dashed consistency links between `cst` and `ast` and the light blue dashed 'vertical correspondence' between `ast` and `ast'`. How these links are obtained is a separable concern, though, and in this chapter we will focus on how to restore consistency assuming the existence of diagonal links. (We will discuss this issue again in Section 5.6.2.)

When it is time to put the modified view back into the source, the links between the source and the modified view are used to guide what regions in the old source should be retained in the new one and at what positions. In addition to the source and view, the *put* function of a retentive lens also takes a collection of links, and provides what we call the *triangular guarantee*, as illustrated in Figure 5.2: when updating `cst` with `ast'`, the region `Neg "a neg" _` (i.e. syntactic sugar negation) connected by the red dashed link is guaranteed to be preserved in the result `cst'` (as opposed to changing it to a `Minus`), and the preserved region will be linked to the same region `Sub (Num 0) _` of `ast'` if we run `getE cst'`. The Retentiveness law will be a formalisation of the triangular guarantee.

5.1.3 Formalisation of Links

We start with some notations.

Relations. Relations from set A to set B are subsets of $A \times B$, and we denote the type of these relations by $A \sim B$. Given a relation $r : A \sim B$, define its *converse* $r^\circ : B \sim A$ by $r^\circ = \{ (b, a) \mid (a, b) \in r \}$, its *left domain* by $\textsc{ldom}\,(r) = \{ a \in A \mid \exists b.\ (a, b) \in r \}$, and its *right domain* by $\textsc{rdom}\,(r) = \textsc{ldom}\,(r^\circ)$. The composition $r \cdot s : A \sim C$ of two relations $r : A \sim B$ and $s : B \sim C$ is defined as usual by $r \cdot s = \{ (a, c) \mid \exists b.\ (a, b) \in r \wedge (b, c) \in s \}$. The type of partial functions from A to B is denoted by $A \nrightarrow B$. The *domain* $\textsc{dom}(f)$ of a function $f : A \nrightarrow B$ is the subset of A on which f is defined; when f is total, i.e. $\textsc{dom}(f) = A$, we write $f : A \to B$. We will allow functions to be implicitly lifted to relations: a function $f : A \nrightarrow B$ also denotes a relation $f : B \sim A$ such that $(f\,x, x) \in f$ for all $x \in \textsc{dom}(f)$[1].

Patterns and Paths. We will work within a universal set *Tree* of trees, which is inductively built from all possible finitely branching constructors. (The semantics of an algebraic data type is then the subset of *Tree* that consists of those trees built with only the constructors of the data type.) Similarly, the set *Pattern* is inductively built from all possible finitely branching constructors,

[1] This flipping of domain and codomain (from $A \nrightarrow B$ to $B \sim A$) makes function composition compatible with relation composition: a function composition $g \circ f$ lifted to a relation is the same as $g \cdot f$, i.e. the composition of g and f as relations.

variables, and a distinguished wildcard element `_`. We will also need a set *Path* of all possible paths for navigating from the root of a tree to one of its subtrees. The exact representation of paths is not crucial: paths are only required to support some standard operations such as $sel : Tree \times Path \nrightarrow Tree$ such that $sel(t, p)$ is the subtree of t at the end of path p (starting from the root), or undefined if p does not exist in t; we will mention these operations in the rest of the paper as the need arises. But, when giving concrete examples, we will use one particular representation: a path is a list of natural numbers indicating which subtree to go into at each node—for instance, starting from the root of `cst` in Figure 5.2, the empty path `[]` points to the root node `Plus`, the path `[0]` points to `"a plus"` (which is the first subtree under the root), and the path `[2,0]` points to `"a neg"`.

Regions and Links. We define a collection of links between two trees as a relation of type $Region \sim Region$, where $Region = Pattern \times Path$: a region is identified by a path leading to a subtree and a pattern describing the part of the subtree included in the region. Briefly, a link is a pair of regions, and a collection of links is a relation between regions of two trees. For brevity we will write *Links* for $Region \sim Region$.

Example 5.1.1. In Figure 5.2, the two shaded regions of `cst` are described by `(Plus "a plus" _ _ , [])` and `(Neg "a neg" _ , [2])` respectively. The diagonal link between `cst` and `ast'` is represented as the link collection `{((Neg "a neg" _ , [2]) , (Sub (Num 0) _ , [0]))}` (despite being a singleton), which can also be regarded as a relation relating `(Neg "a neg" _ , [2])` to `(Sub (Num 0) _ , [0])` and vice versa.

An arbitrary collection of links may not make sense for a given pair of trees though—a region mentioned by some link may not exist in the trees at all. We should therefore characterise when a collection of links is valid for two trees.

Definition 5.1.2 (Region Containment)**.** For a tree t and a set of regions $\Phi \subseteq Region$, we say that $t \models \Phi$ (read 't contains Φ') exactly when

$$\forall (pat, path) \in \Phi. \quad sel(t, path) \text{ matches } pat.$$

Definition 5.1.3 (Valid Links)**.** Given $ls : Links$ and two trees t and u, we say that ls is *valid* for t and u, denoted by $t \overset{ls}{\longleftrightarrow} u$, exactly when

$$t \models \textsc{ldom}\,(ls) \quad \text{and} \quad u \models \textsc{rdom}\,(ls).$$

Example 5.1.4. The link collection given in Example 5.1.1 is valid for `cst` and `ast'` because `cst` $\models$ `{(Neg "a neg" _ , [2])}`, `ast'` $\models$ `{(Sub (Num 0) _ , [0])}`.

5.1.4 Formalisation of Retentive Lenses

Now we have all the ingredients for the formal definition of retentive lenses.

Definition 5.1.5 (Retentive Lenses)**.** For a set S of source trees and a set V of view trees, a retentive lens between S and V is a pair of functions

$$\begin{aligned} \mathit{get} &: S \nrightarrow V \times \mathit{Links} \\ \mathit{put} &: S \times V \times \mathit{Links} \nrightarrow S \end{aligned}$$

satisfying

- *Hippocraticness:* if $\mathit{get}\ s = (v, ls)$, then $(s, v, ls) \in \textsc{dom}(\mathit{put})$ and

$$\mathit{put}\ (s, v, ls) = s\ ; \tag{5.1}$$

- *Correctness:* if $\mathit{put}\ (s, v, ls) = s'$, then $s' \in \textsc{dom}(\mathit{get})$ and

$$\mathit{get}\ s' = (v, ls') \quad \text{for some } ls'\ ; \tag{5.2}$$

- *Retentiveness:*

$$\mathit{fst} \cdot ls \subseteq \mathit{fst} \cdot ls' \tag{5.3}$$

  where $\mathit{fst} : A \sim A \times B$ is the first projection function (lifted to a relation).

Modulo the handling of links, Hippocraticness and Correctness remain the same as their original forms (in the definition of well-behaved lenses). Retentiveness further states that the input links ls must be preserved, except for the location of source regions (i.e. $\textsc{rdom}\,(\mathit{snd} \cdot ls)$ in the compact relational notation). The region patterns (data) and the location of the view region, which are $\mathit{fst} \cdot ls$ in the relational notation, must be exactly the same. Retentiveness formalises the triangular guarantee in a compact way, and we can expand it pointwise to see that it indeed specialises to the triangular guarantee.

Proposition 5.1.6 (Triangular Guarantee)**.** Given a retentive lens, suppose $put\ (s, v, ls) = s'$ and $get\ s' = (v, ls')$. If $((spat, spath), (vpat, vpath)) \in ls$, then for some $spath'$ we have $s' \models \{(spat, spath')\}$ and $((spat, spath'), (vpat, vpath)) \in ls'$.

Example 5.1.7. In Figure 5.2, if the `put` function takes `cst`, `ast'`, and links `ls` = `{((Neg "a neg" _ , [2]) , (Sub (Num 0) _ , [0]))}` as arguments and successfully produces an updated source `s'`, then `get s'` will succeed. Let `(v,ls') = get s'`; we know that we can find a link in `ls'` with the path of its source region removed: `c = (Neg "a neg" _ , (Sub (Num 0) _ , [0])) ∈ fst · ls'`. So the view region referred to by `c` is indeed the same as the one referred to by the input link, and having `c ∈ fst · ls'` means that the region in `s'` corresponding to the view region will match the pattern `Neg "a neg" _`.

Finally, we note that retentive lenses are an extension of well-behaved lenses: every well-behaved lens between trees can be directly turned into a retentive lens (albeit in a trivial way).

Example 5.1.8 (Well-behaved Lenses are Retentive Lenses)**.** Given a well-behaved lens defined by $g : S \to V$ and $p : S \times V \to S$, we define $get : S \nrightarrow V \times Links$ and $put : S \times V \times Links \nrightarrow S$ as follows:

$$\begin{array}{lll} get\ \ s & = & (g\ s, \emptyset) \\ put\ \ (s, v, ls) & = & p\ (s, v) \end{array}$$

In the definition, $\textsc{dom}(put)$ is restricted to $\{(s, v, \emptyset)\}$. Hippocraticness and Correctness hold because the underlying g and p are well-behaved. Retentiveness is satisfied vacuously since the input link of put is empty.

5.1.5 Composition of Retentive Lenses

It is standard to provide a composition operator for composing large lenses from small ones. Here we discuss this operator for retentive lenses, which basically follows the definition of composition for well-behaved lenses, except that we need to deal with links carefully.

We use $(\cdot)$ to denote *link composition* and its (overloaded) lifted version that works on two collections of links. (Although $(\cdot)$ was used for relation composition before, we have shown that a single link can be lifted to a singleton link collection and a collection of links can be viewed as a relation; so it is reasonable to reuse the same symbol.) Below we use $lens_{AB}$ to denote a retentive lens that synchronises trees of sets A and B, get_{AB} and put_{AB} the get and put functions of the lens, l_{ab} a link between tree a (of set A) and tree b (of set B), and ls_{ab} a collection of links between a and b.

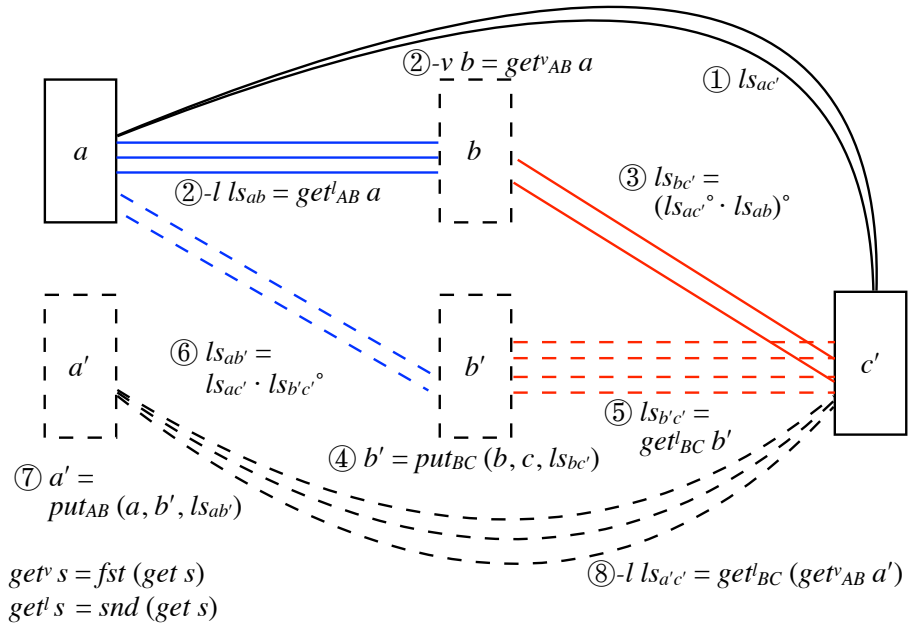


Figure 5.3: The *put* behaviour of a composite retentive lens $lens_{AB}; lens_{BC}$. (The process of *put* is divided into steps ① to ⑧. Step ⑧ produces consistency links for showing triangular guarantee.)

Definition 5.1.9 (Retentive Lens Composition)**.** Given two retentive lenses $lens_{AB}$ and $lens_{BC}$, define the *get* and *put* functions of their composition by

$$
\begin{aligned}
&get_{AC}\ a = (c, ls_{ab} \cdot ls_{bc}) \\
&\quad where\ (b, ls_{ab}) = get_{AB}\ a \\
&\qquad\ (c, ls_{bc}) = get_{BC}\ b
\end{aligned}
\qquad\qquad
\begin{aligned}
&put_{AC}\ (a, c', ls_{ac'}) = a' \\
&\quad where\ (b, ls_{ab}) = get_{AB}\ a \\
&\qquad ls_{bc'} \quad = (ls^{\circ}_{ac'} \cdot ls_{ab})^{\circ} \\
&\qquad b' \quad = put_{BC}\ (b, c', ls_{bc'}) \\
&\qquad ls_{b'c'} \quad = fst\ (get_{BC}\ b') \\
&\qquad ls_{ab'} \quad = ls_{ac'} \cdot ls^{\circ}_{b'c'} \\
&\qquad a' \quad = put_{AB}\ (a, b', ls_{ab'})
\end{aligned}
$$

where get_{AB} and put_{AB} (resp. get_{BC} and put_{BC}) are the corresponding *get* and *put* functions of $lens_{AB}$ (resp. $lens_{BC}$), and ls° is the collection of links produced from ls by swapping each link's starting point and ending point. (Recall that a collection of links is regarded as a relation.)

The *get* behaviour of a composite retentive lens is straightforward; the *put* behaviour, on the other hand, is a little complex and can be best understood with the help of Figure 5.3. Let us first

```
Expr <---> Arith                    Term <---> Arith
  Plus   _ x y   ~   Add x y          Lit    _  i   ~   Num i
  Minus  _ x y   ~   Sub x y          Neg    _  r   ~   Sub (Num 0) r
  FromT  _ t     ~   t                Paren  _  e   ~   e
```

Figure 5.4: The program in our DSL for synchronising data types defined in Figure 5.1.

recap the composite behaviour of *put* of traditional lenses: in Figure 5.3, if we need to propagate changes from data c' back to data a without links, we will first construct the *intermediate* data b (by running $get_{AB}\ a$), propagate changes from c' to b and produce b', and finally use b' to update a. The composition of retentive lenses is similar: besides the intermediate data b, we also need to construct intermediate links $ls_{bc'}$ (③ in the figure) for retaining information when updating b to b', so that we can further construct intermediate links $ls_{ab'}$ (⑥ in the figure) for retaining information when updating a to a' using b'.

Theorem 5.1.10 (Composability)**.** The composite lens $lens_{AC} = lens_{AB}; lens_{BC}$ from two retentive lenses $lens_{AB}$ and $lens_{BC}$ is still a retentive lens.

The proof is available in the Appendix A.1.

5.2 A DSL for Retentive Tree Transformation

The definition of retentive lenses is somewhat complex, but we can ease the task of constructing retentive lenses with a declarative domain-specific language. Our DSL is designed to describe consistency relations between algebraic data types, and from each consistency relation defined in the DSL, we can obtain a pair of *get* and *put* functions forming a retentive lens. Below we will give an overview of the DSL and how retentive lenses are derived from programs in the DSL using the arithmetic expression example (Section 5.2.1), the syntax (Section 5.2.2) and semantics (Section 5.2.3) of the DSL, and finally the theorem stating that the generated lenses satisfy the required laws (Theorem 5.2.1). Proof of the theorem is in Appendix A.2, but the essence is given in the last part of Section 5.2.1.

5.2.1 Overview of the DSL

In this subsection, we introduce our DSL by describing the consistency relations between the concrete syntax and abstract syntax of the arithmetic expression example in Figure 5.1. Furthermore, we show how to flexibly update the `cst` in Figure 5.2 in different ways with different input links.

In our DSL, we define data types in HASKELL syntax and describe consistency relations between them that bear some similarity to *get* functions. For example, the data type definitions for `Expr` and `Term` written in our DSL remain the same as those in Figure 5.1, and the consistency relations between them (i.e. `getE` and `getT` in Figure 5.1) are expressed as the ones in Figure 5.4. Here we specify two consistency relations similar to `getE` and `getT`: one between `Expr` and `Arith`, and the other between `Term` and `Arith`. Each consistency relation is further defined by a set of *inductive rules*, stating that if the subtrees matched by the same variable appearing on the left-hand side (i.e. source side) and right-hand side (i.e. view side) are consistent, then the larger pair of trees constructed from these subtrees are also consistent. (For primitive types which do not have constructors such as integers and strings, we consider them consistent if and only if they are equal.) Take

```
Plus _ x y ~ Add x y
```

for example: it means that if x_s is consistent with x_v, and y_s is consistent with y_v, then `Plus` a x_s y_s and `Add` x_v y_v are consistent for any value a, where a corresponds to the 'don't-care' wildcard in `Plus _ x y`. So the meaning of `Plus _ x y ~ Add x y` can be better understood as the following proof rule:

$$\frac{x_s \sim x_v \quad y_s \sim y_v}{\texttt{Plus}\ a\ x_s\ y_s \sim \texttt{Add}\ x_v\ y_v}$$

Each consistency relation is translated to a pair of *get* and *put* functions defined by case analysis generated from the inductive rules. Detail of the translation will be given in Section 5.2.3, but the idea behind the translation is a fairly simple one which establishes Retentiveness by construction. For *get*, the rules themselves are already close to function definitions by pattern matching, so what we need to add is only the computation of output links. For *put*, we use the rules backwards and define a function that turns the regions of an input view into the regions of the new source, reusing regions of the old source wherever required: when there is an input link connected to the current view region, *put* grabs the source region at the other end of the link in the old source; otherwise, *put* creates a new source region as described by the left-hand side of an appropriate rule.

For example, suppose that the *get* and *put* functions generated from the consistency relation `Expr <---> Arith` are named `getEA` and `putEA` respectively. The inductive rule `Plus _ x y ~ Add x y` generates the definition for `getEA s` when `s` matches `Plus _ x y`: `getEA (Plus _ x y)` computes a view recursively in the same way as `getE` in Figure 5.1; furthermore, it produces a new link between the top regions `Plus` and `Add`, and keeps the links produced by the recursive

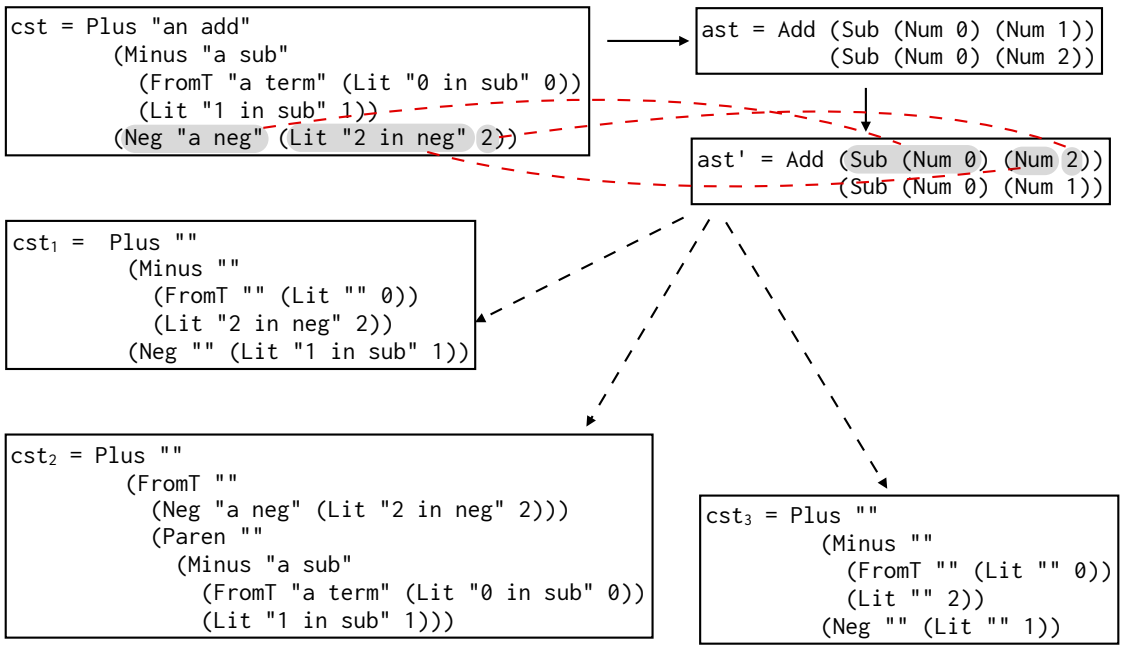


Figure 5.5: `cst` is updated in many ways. (Red dashed lines are some of the input links for $\texttt{cst}_2$.)

calls `getEA x` and `getEA y`. In the *put* direction, the inductive rule `Plus _ x y ~ Add x y` leads to a case `putEA s (Add x y) ls`, under which there are two subcases: if there is any link in `ls` that is connected to the `Add` region at the top of the view, `putEA` grabs the region at the other end of the link in the old source and tries to use it as the top part of the new source; if such a link does not exist, `putEA` uses a `Plus` with a default annotation as a substitute for the top part of the new source. In either case, the subtrees of the new source at the positions marked by `x` and `y` are computed recursively from the view subtrees `x` and `y`.

With `getEA` and `putEA`[1] generated from Figure 5.4, we can synchronise the (old) `cst` and (modified) `ast'` in different ways by running `putEA` with different input links. Figure 5.5 illustrates this:

- If the user assumes that the two non-zero numbers `Num 1` and `Num 2` in `ast` are exchanged, then four links (between `cst` and `ast'`) should be established: a link connects region `Lit "1 in sub" _` with region `Num _` (of the tree `Num 1`); a link connects `Lit "2 in neg" _` with `Num _` (of the tree `Num 2`); two links connects `1` and `1`, and `2` and `2`. The result is $\texttt{cst}_1$, where

[1]There are also `getTA` and `putTA` generated from the consistency relation `Term <---> Arith`, which are mutually recursive with `getEA` and `putEA` respectively.

the literals `1` and `2` are swapped, along with their annotations. Note that all other annotations disappeared because the user did not include links for preserving them.

- If the user thinks that the two subtrees of `Add` are swapped and passes `putEA` links that connect not only the roots of the subtrees of `Plus` and `Add` but also all their inner subtrees, the desired result should be $\texttt{cst}_2$, which represents '`-2 + (0 - 1)`' and retains all annotations, in effect swapping the two subtrees under `Plus` and adding two constructors `FromT` and `Paren` to satisfy the type constraint. Figure 5.5 shows the input links for `Neg` and its subtrees.
- If the user believes that `ast'` is created from scratch and not related to `ast`, then $\texttt{cst}_3$ may be the best choice, which represents the same expression as $\texttt{cst}_1$ except that all the annotations are removed.

While the core idea is simple, there are cases in which the translated functions do not constitute valid retentive lenses, and the crux of Theorem 5.2.1 is finding suitable ways of computation or reasonable conditions to circumvent all such cases (some of which are rather subtle). The following cases should give a good idea of what is involved in the correctness of the theorem.

I. *The translated functions may not be well-defined.* For example, in the *get* direction, an arbitrary set of rules may assign zero or more than one view to a source, making *get* partial (which, though allowed by the definition, we want to avoid) or ill-defined, and we will impose (fairly standard) restrictions on patterns to preclude such rules. These restrictions are sufficient to guarantee that exactly one rule is applicable in the *get* direction but not in the *put* direction, in which we need to carefully choose a rule among the applicable ones or risk non-termination (e.g. producing an infinite number of parentheses by alternating between the `Paren` and `FromT` rules).

II. *A region grabbed by put from the old source may not have the right type.* For example, if *put* is run on `cst`, `ast'`, and the link between them in Figure 5.2, it has to grab the source region `Reg "a neg" _`, which has type `Term`, and install it as the second argument of `Plus`, which has to be of type `Expr`. In this case there is a way out since we can convert a `Term` to an `Expr` by wrapping the `Term` in the `FromT` constructor. We will formulate conditions under which such conversions are needed and can be synthesised automatically.

III. *Hippocraticness may be accidentally invalidated by put.* Suppose that there is another parenthesis constructor `Brac` that has the same type as `Paren` and for which a similar rule `Brac _ e ~ e` is supplied. Given a source that starts with `Brac "" (Paren "" ...)`, *get* will produce two

links (among others) relating both the `Brac` and `Paren` regions with the empty region at the top of the view. If *put* is immediately invoked on the same source, view, and links, it may choose to process the link attached to the `Paren` region first rather than the one attached to the `Brac` region, so that the new source starts with `Paren "" (Brac "" ...)`, invalidating Hippocraticness. Therefore *put* has to carefully process the links in the right order for Hippocraticness to hold.

IV. *Retentiveness may be invalidated if put does not correctly reject invalid input links.* Unlike *get*, which can easily be made total, *put* is inherently partial since input links may well be invalid and make Retentiveness impossible to hold. For example, if there is an input link relating a `Neg` region and an `Add` region, then it is impossible for *put* to produce a result that satisfies Retentiveness since *get* does not produce a link of this form. Instead, *put* must correctly reject invalid links for Retentiveness to hold. Apart from checking that input links have the right forms as specified by the rules, there are more subtle cases where the view regions referred to by a set of input links are overlapping—for example, in a view starting with `Sub (Num 0) ...` there can be links referring to both the `Sub _ _` region and the `Sub (Num 0) _` region at the top. Our *get* cannot produce overlapping view regions, and therefore such input links must be detected and rejected as well.

In the last part of this subsection, we show that the DSL is possible to handle general tree transformations despite the fact that it is tailored for describing consistency relations between syntax trees. The following are some small but typical programming examples.

Let us consider the binary trees

```
data BinT a = Tip | Node a (BinT a) (BinT a) .
```

We can concisely define the *mirror* consistency relation between a tree and its mirroring as

```
BinT Int <---> BinT Int
  Tip          ~  Tip
  Node i x y   ~  Node i y x .
```

We demonstrate the implicit use of some other consistency relations when defining a new one. Suppose that we have defined the following consistency relation between natural numbers and boolean values:

```
Nat <---> Bool
  Succ _  ~  True
  Zero    ~  False .
```

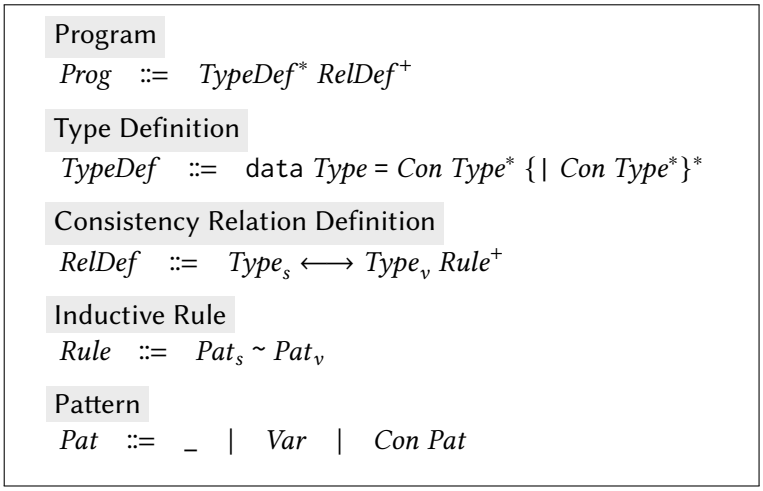


Figure 5.6: Syntax of the DSL.

Then we can easily describe the consistency relation between a binary tree over natural numbers and a binary tree over boolean values:

```
BinT Nat  <---> BinT Bool
  Tip           ~  Tip
  Node x ls rs  ~  Node x ls rs .
```

Let us consider rose trees, a data structure mutually defined with lists:

```
data RTree a = RNode a (List (RTree a))
data List  a = Nil | Cons a (List a) .
```

We can define the following consistency relation to associate the left spine of a tree with a list:

```
RTree Int <---> List Int
  RNode i Nil         ~  Cons i Nil
  RNode i (Cons x _)  ~  Cons i x .
```

5.2.2 Syntax

The syntax of our DSL is summarised in Figure 5.6, where nonterminals are in *italic*; terminals are typeset in `typewriter` font; {} is for grouping; $^?$, *, and $^+$ represent zero-or-one occurrence, zero-or-more occurrence, and one-or-more occurrence respectively, and *Type*, *Con*, and *Var* are syntactic categories (whose definitions are omitted) for the names of types, constructors, and

variables respectively. We sometimes additionally attach a subscript s or v to a symbol to mean that the symbol is related to sources or views. A program consists of two parts: data types definitions and consistency relations between these data types. We adopt the Haskell syntax for data type definitions—a data type is defined by specifying a set of data constructors and their argument types. As for the definitions of consistency relations, each of them starts with $Type_s \leftrightarrow Type_v$, declaring the source and view types for the relation. The body of each consistency relation is a list of inductive rules, each of which defined by a pair of source and view patterns $Pat_s \sim Pat_v$, where a pattern can include wildcards, variables, and constructors.

5.2.2.1 Syntactic Restrictions

We impose some syntactic restrictions to guarantee that programs in our DSL indeed give rise to retentive lenses (Theorem 5.2.1).

On *patterns*, we require (i) pattern coverage: for any consistency relation $S \leftrightarrow V = \{ p_i \sim q_i \mid 1 \leqslant i \leqslant n \}$ defined in a program, $\{ p_i \}$ should cover all possible cases of type S, and $\{ q_i \}$ should cover all cases of type V. We also require (ii) source pattern disjointness: any distinct p_i and p_j should not be matched by the same tree. Finally, (iii) a bare variable pattern is not allowed on the source side (e.g. $x \sim D\ x$), and (iv) wildcards are not allowed on the view side (e.g. $C\ x \sim D\ \_\ x$), and (v) the source side and the view side must use exactly the same set of variables. These conditions ensure that *get* is total and well-defined (ruling out Case I in Section 5.2.1).

To state the next requirement we need a definition: two data types S_1 and S_2 defined in a program are *interchangeable* in data type S exactly when (i) there are some data type V' and V for which consistency relations $S_1 \leftrightarrow V'$, $S_2 \leftrightarrow V'$ and $S \leftrightarrow V$ are defined in the program, and (ii) S may have subterms of type S_1 and S_2, and V may have subterms of type V'. If S_1 and S_2 are interchangeable, then Case II (Section 5.2.1) may happen: when doing *put* on S and V there might be input links dictating that values of type S_2 should be retained in a context where values of type S_1 are expected, or vice versa. When this happens, we need two-way conversions between S_1 and S_2.

We choose a simple way to ensure the existence of conversions: for any interchangeable types S_1 and S_2 with $S_1 \leftrightarrow V'$ and $S_2 \leftrightarrow V'$ defined, we require that there exists a sequence of data types in the program

$$S_1 = T_1,\ T_2,\ \cdots,\ T_{n-1},\ T_n = S_2$$

with $n >= 2$ such that for any $1 \leqslant i < n$, consistency relation $T_i \leftrightarrow V'$ is defined and has a rule $Pat_i \sim x$ whose source pattern Pat_i contains exactly one variable, and its type in Pat_i is T_{i+1} (we also

require such a sequence with the roles of S_1 and S_2 switched). With rule $Pat_i \sim x$, we immediately get a function $t_i : T_{i+1} \to T_i$ contructing a T_i from a term v of T_{i+1} by substituting v for x in Pat_i (and filling wildcard positions with default values). Then we have the needed conversion function:

$$inj_{S_2 \to S_1 @ V'} = t_{n-1} \circ \cdots \circ t_2 \circ t_1 \tag{5.4}$$

(and similary $inj_{S_1 \to S_2 @ V'}$). For example, `FromT _ t ~ t` gives rise to a function

$$\texttt{inj}_{\texttt{Term} \to \texttt{Expr@Arith}}\ x\ =\ \texttt{FromT ""}\ x$$

and it can be used to convert `Term` to `Expr` whenever needed when doing *put* with view type `Arith`.

5.2.3 Semantics

We give the semantics of our DSL in terms of a translation into 'pseudo-Haskell', where we may replace chunks of Haskell code with natural language descriptions to improve readability. As in Section 5.1.4, let *Tree* be the set of values of any algebraic data type, and *Pattern* the set of all patterns. For a pattern $p \in Pattern$, $Vars\ p$ denotes the set of variables in p. For each $v \in Vars\ p$, $TypeOf\ (p, v)$ is (the set of all values of) the type of v in pattern p, and $path\ (p, v)$ is the path of variable v in pattern p. We use the following functions (two of which are dependently typed) to manipulate patterns:

$$\begin{aligned}
isMatch &: \quad Pattern \times Tree \to Bool \\
decompose &: (p \in Pattern) \times Tree \nrightarrow \big(Vars\ p \to Tree\big) \\
reconstruct &: (p \in Pattern) \times \big(Vars\ p \to Tree\big) \nrightarrow Tree \\
fillWildcards &: \quad Pattern \times Tree \nrightarrow Pattern \\
fillWildcardsWD &: \quad Pattern \to Pattern \\
eraseVars &: \quad Pattern \to Pattern\ .
\end{aligned}$$

Given a pattern p and a tree t, $isMatch\ (p, t)$ tests whether t matches p. If the match succeeds, $decompose\ (p, t)$ returns a function mapping every variable in p to its corresponding matched subtree of t. Conversely, $reconstruct\ (p, f)$ produces a tree matching p by replacing every occurrence of $v \in Vars\ p$ in p with $f\ v$, provided that p does not contain any wildcard. To remove wildcards, we can use $fillWildcards\ (p, t)$ to replace all the wildcards in p with the corresponding subtrees of t

(coerced into patterns) when t matches p, or use *fillWildcardsWD* to replace all the wildcards with the default values of their types. Finally, *eraseVars* p replaces all the variables in p with wildcards. The definitions of these functions are straightforward and omitted here[1].

5.2.3.1 Get Semantics

For a consistency relation $S \leftrightarrow V$ defined in our DSL with a set of inductive rules $R = \{\, spat_k \sim vpat_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding get_{SV} function has the following type:

$$get_{SV} : S \longrightarrow V \times Links$$

The idea of computing $get\ s$ is to use a rule $spat_k \sim vpat_k \in R$ such that s matches $spat_k$—the restrictions on patterns imply that such a rule uniquely exists for all s—to generate the top portion of the view with $vpat_k$, and then recursively generate subtrees for all variables in $spat_k$. The *get* function also creates links in the recursive procedure: when a rule $spat_k \sim vpat_k \in R$ is used, it creates a link relating the matched parts/regions in the source and view, and extends the paths in the recursively computed links between the subtrees. In all, the *get* function defined by R is:

$$\begin{aligned} get_{SV}\ s = {} & (reconstruct\ (vpat_k, fst \circ vls),\ l_{root} \cup links) \\ & \textbf{where}\ \text{find } k \text{ such that } spat_k \sim vpat_k \in R \text{ and } isMatch(spat_k, s) \\ & \quad vls = (get \circ decompose\ (spat_k, s)) \in Vars\ spat_k \longrightarrow V \times Links \\ & \quad spat' = eraseVars\ (fillWildcards\ (spat_k, s)) \\ & \quad l_{root} = \{\, ((spat', [\,]), (eraseVars\ vpat_k, [\,])) \,\} \\ & \quad links = \big\{\, ((spat, path\ (spat_k, v) +\!\!+ spath), (vpat, path\ (vpat_k, v) +\!\!+ vpath)) \\ & \qquad\qquad \mid v \in Vars\ vpat_k, ((spat, spath), (vpat, vpath)) \in snd\ (vls\ v) \,\big\}\ . \end{aligned} \tag{5.5}$$

The auxiliary function $path : (p \in Pattern) \times Vars\ p \longrightarrow Path$ returns the path from the root of a pattern to one of its variables, and $(+\!\!+)$ is path concatenation. While the recursive call is written as $get \circ decompose\ (spat_k, s)$ in the definition above, to be precise, *get* should have different subscripts $TypeOf\ (spat_k, v)$ and $TypeOf\ (vpat_k, v)$ for different $v \in Vars\ spat_k$.

[1] Non-linear patterns are allowed: multiple occurrences of the same variable in a pattern must have the same type and they must capture the same value.

5.2.3.2 Put Semantics

For a consistency relation $S \leftrightarrow V$ defined in our DSL as $R = \{\, spat_k \sim vpat_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding put_{SV} function has the following type:

$$put_{SV} : Tree \times V \times Links \nrightarrow S\,.$$

The source argument of *put* is given the generic type *Tree* since the type of the old source may be different from the type of the result that *put* is supposed to produce. Given arguments (s, v, ls), *put* is defined by two cases depending on whether the root of the view is within a region referred to by the input links, i.e. whether there is some $(\_, (\_, [\,])) \in ls$.

- In the first case where the root of the view is not within any region of the input links, *put* selects a rule $spat_k \sim vpat_k \in R$ whose $vpat_k$ matches v—our restriction on view patterns implies that at least one such rule exists for all v—and uses $spat_k$ to build the top portion of the new source: wildcards in $spat_k$ are filled with default values and variables in $spat_k$ are filled with trees recursively constructed from their corresponding parts of the view.

$$\begin{aligned} put_{SV}\ (s, v, ls) &= reconstruct\ (spat'_k, ss) && (5.6)\\ &\quad \textbf{where } \text{find } k \text{ such that } spat_k \sim vpat_k \in R \text{ and } isMatch\ (vpat_k, v)\\ &\qquad\qquad \text{and } k \text{ satisfies the extra condition below}\\ &\quad vs = decompose\ (vpat_k, v)\\ &\quad ss = \lambda\,(t \in Vars\ spat_k) \longrightarrow\\ &\qquad put(s, vs\ t, divide\ (path\ (vpat_k, t)), ls) && (5.7)\\ &\quad spat'_k = fillWildcardsWD\ spat_k\\ divide\ (prefix, ls) &= \{\, (r_s, (vpat, vpath)) \mid (r_s, (vpat, prefix +\!\!+ vpath) \in ls) \,\} && (5.8) \end{aligned}$$

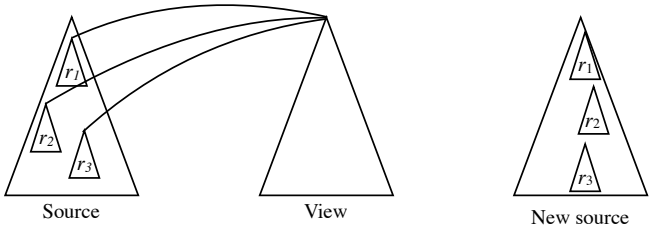
The omitted subscripts of *put* in (5.7) are $TypeOf\ (spat_k, t)$ and $TypeOf\ (vpat_k, t)$. Additionally, if there is more than one rule whose view pattern matches v, the first rule whose view pattern is *not* a bare variable pattern is preferred for avoiding infinite recursive calls: if $vpat_k = x$, the size of the input of the recursive call in (5.7) does not decrease because $vs\ t = v$ and $path\ (t, vpat_k) = [\,]$. For example, when the view patterns of both `Plus _ x y ~ Add x y` and `FromT _ t ~ t` match a view tree, the former is preferred. This helps to avoid non-termination of *put* as mentioned in Case I in (the last part of) Section 5.2.1.

- In the case where the root of the view is an endpoint of some link, *put* uses the source region (pattern) of the link as the top portion of the new source.

$$put_{SV}\ (s, v, ls) = inj_{TypeOf\ spat_k \to S@V}(reconstruct(spat'_k, ss)) \tag{5.9}$$

$$\begin{aligned}
&\textbf{where } l = ((spat, spath), (vpat, vpath)) \in ls \\
&\qquad \text{such that } vpath = [\,],\ spath \text{ is the shortest} \\
&\quad \text{find } k \text{ such that } spat_k \sim vpat_k \in R \text{ and } spat \\
&\qquad \text{is } eraseVars\ (fillWildcards\ (spat_k, t)) \text{ for some } t \\
&\quad spat'_k = fillWildcards\ (spat_k, spat) \\
&\quad vs = decompose\ (vpat_k, v) \\
&\quad ss = \lambda\,(t \in Vars\ spat_k) \longrightarrow \\
&\qquad put(s, vs\ t, divide\ (path\ (vpat_k, t)), ls \setminus \{\, l \,\}))
\end{aligned} \tag{5.10}$$

When there is more than one source region linked to the root of the view, to avoid Case III in Section 5.2.1, *put* chooses the source region whose path is the shortest, which ensures that the preserved region patterns in the new source will have the same relative positions as those in the old source, as the following figure shows.



Since the linked source region (pattern) does not necessarily have type S, we need to use the function $inj_{TypeOf\ spat_k \to S}@V$ (Equation 5.4) to convert it to type S; this function is available due to our requirement on interchangeable data types (see Syntax Restrictions in Section 5.2.2).

5.2.3.3 Domain of *put*

To avoid Case IV in Section 5.2.1, in the actual implementation of *put* there are runtime checks for detecting invalid input links, but these checks are omitted in the above definition of *put* for clarity.

We extract these checks into a separate function *check* below, which also serves as a decision procedure for the domain of *put*.

$$check : Tree \times V \times Links \longrightarrow Bool$$

$$check\ (s, v, ls) = \begin{cases} chkWithLink\ (s, v, ls) & \text{if some } ((\_, \_), (\_, [\,])) \in ls \\ chkNoLink\ (s, v, ls) & \text{otherwise} \end{cases}$$

chkNoLink corresponds to the first case of *put* (5.6).

$$chkNoLink\ (s, v, ls) = cond_1 \wedge cond_2 \wedge cond_3$$

where find k such that $spat_k \sim vpat_k \in R$ and $isMatch\ (vpat_k, v)$
and k satisfies the same condition as in (5.6)

$$vs = decompose\ (vpat_k, v)$$

$$vp\ t = path\ (vpat_k, t)$$

$$cond_1 = ls == \left(\bigcup_{t \in Vars\ spat_k} addVPrefix\ (vp\ t, divide\ (vp\ t, ls)) \right)$$

$$cond_2 = \bigwedge_{t \in Vars\ spat_k} check\ (s, vs\ t, divide\ (vp\ t, ls))$$

$cond_3$ = **if** $vpat_k$ is some bare variable pattern 'x' **then**
$TypeOf\ (spat_k, x) \leftrightarrow V$ has a rule $spat_j \sim vpat_j$ such that
$isMatch\ (vpat_j, v)$ and $vpat_j$ is not a bare variable pattern

$$addVPrefix\ (prefix, rs) = \{\, ((a, b), (c, prefix +\!\!+ d)) \mid ((a, b), (c, d)) \in rs \,\}$$

The *divide* function is defined as in Equation 5.8. Condition $cond_1$ checks that every link in ls is processed in one of the recursive calls, i.e. the path of every view region of ls starts with $path\ (vpat_k, t)$ for some t. (Specifically, if $Vars\ spat_k$ is empty, ls in $cond_1$ should also be empty meaning that all the links have already been processed.) $cond_2$ summarises the results of *check* for recursive calls. $cond_3$ guarantees the termination of recursion: When $vpat_k$ is a bare variable pattern, the recursive call in Equation 5.7 does not decrease the size of any of its arguments; $cond_3$ makes sure that such non-decreasing recursion will not happen in the next round[1] for avoiding

[1] For presentation purposes we only check two rounds here, but in general we should check $N + 1$ rounds where N is the number data types defined in the program.

infinite recursive calls.

For *chkWithLink*, as in the corresponding case of *put* (Equation 5.9), let $l = ((spat, spath), (vpat, vpath)) \in ls$ such that $vpath = [\,]$ and *spath* is the shortest when there is more than one such link.

$$
\begin{aligned}
&chkWithLink\ (s, v, ls) = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \\
&\quad \textbf{where} \\
&cond_1 = isMatch\ (spat, sel\ (s, spath)) \wedge isMatch\ (vpat, sel\ (v, vpath)) \\
&cond_2 = \exists!(spat_k, vpat_k) \in R.\ vpat = eraseVars\ vpat_k \\
&\qquad\qquad \wedge\ spat \text{ is } eraseVars\ (fillWildcards\ (spat_k, t)) \text{ for some } t \\
&cond_3 = ls == (\{\, l \,\} \cup \bigcup_{t \in Vars\ spat_k} addVPrefix\ (path\ (vpat_k, t), \\
&\qquad\qquad\qquad\qquad divide\ (path\ (vpat_k, t), ls \setminus \{\, l \,\}))) \\
&cond_4 = \bigwedge_{t \in Vars\ spat_k} check\ (s, vs\ t, divide\ (path\ (vpat_k, t)), ls \setminus \{\, l \,\}))
\end{aligned}
$$

$cond_1$ makes sure that the link l is valid (Definition 5.1.3) and $cond_2$ further checks that it can be generated from some rule of the consistency relations. $cond_3$ and $cond_4$ are for recursive calls: the latter summarises the results for the subtrees and the former guarantees that no link will be missed. It is $cond_3$ that rejects the subtle case of overlapping view regions as described at the end of Case IV in Section 5.2.1.

5.2.3.4 Retentiveness of the DSL

We can now state our main theorem in terms of the definitions of *get* and *put* above.

Theorem 5.2.1. [Generated Lenses are Retentive] Let *put′* be *put* with its domain intersected with $S \times V \times Links$, *get* and *put′* form a retentive lens as in Definition 5.1.5.

The proof goes by induction on the size of the arguments to *put* or *get* and can be found in Appendix A.2.

5.3 Edit Operations and Link Maintenance

In the retentive lens framework, a *get* function only produces horizontal links between a source and its consistent view, while the input links to a *put* function are the ones between a source and

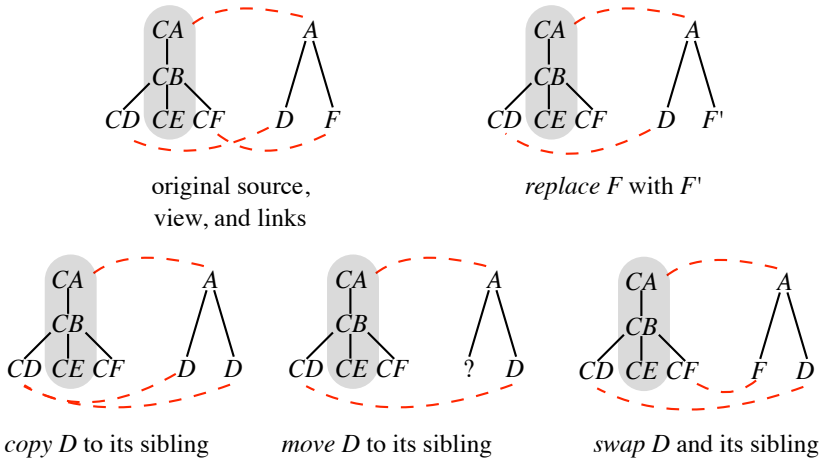


Figure 5.7: How edit operations *replace*, *copy*, *move*, and *swap* main links.

a modified view. To bridge the gap, in this section, we demonstrate how to update the view while maintaining the links using a set of typical *edit operations* (on views). These edit operations will be used in the three case studies in the next section.

We define four edit operations, *replace*, *copy*, *move*, and *swap*, of which *move* and *swap* are defined in terms of *copy* and *replace*. The edit operations accept not only an AST but also a set of links, which is updated along with the AST. The interface has been designed in a way that the last argument of an edit operation is the pair of the AST and links, so that the user can use Haskell's ordinary function composition to compose a sequence of edits (partially applied to all the other arguments). The implementation of the four edit operations takes less than 40 lines of Haskell code, as our DSL already generates useful auxiliary functions such as fetching a subtree according to a path in some tree.

We briefly explain how the edit operations update links, as illustrated in Figure 5.7: Replacing a subtree at path p will destroy all the links previously connecting to path p. Copying a subtree from path p to path p' will duplicate the set of links previously connecting to p and redirect the duplicated links to connect to p'. Moving a subtree from p to p' will destroy links connecting to p' and redirect the links (previously) connecting to p to connect to p'. Swapping subtrees at p and p' will also swap the links connecting to p and p'.

5.4 Case Studies

We demonstrate how our DSL works for the problems of code refactoring (Fowler and Beck, 1999), resugaring (Pombrio and Krishnamurthi, 2014, 2015), and XML synchronisation (Pacheco et al., 2014b), all of which require that we constantly make modifications to ASTs and synchronise them with CSTs. For all these problems, retentive lenses provide a systematic way for the user to preserve information of interest in the original CST after synchronisation.

5.4.1 Refactoring

As we will report below, we have programmed the consistency relations between CSTs and ASTs for a small subset of Java 8 (Gosling et al., 2014) and tested the generated retentive lens on a particular refactoring. Even though the case study is small, we believe that our framework is general enough: We have surveyed the standard set of refactoring operations for Java 8 provided by Eclipse Oxygen (with Java Development Tools) and found that all the 23 refactoring operations can be represented as the combinations of our edit operations defined in Section 5.3. Note that position-wise replacement and list alignment (which will be discussed in the last paragraph in Section 5.4.3 and in more detail in Section 5.5.1) is sufficient for only about half of the refactoring operations, in which the code (precisely, subtrees of an AST) is moved around within a list-like structure (such as an expression list or a statement list). For the remaining half of the operations, the code is moved out of its original list-like structure so that some 'global tracking information' such as links are required, where our retentive lenses outperform well-behaved lenses. A summary can be found in Appendix B.

5.4.1.1 The *Push-Down* Code Refactoring

An example of the *push-down* code refactoring is illustrated in Figure 5.8. At first, the user designed a `Vehicle` class and thought that it should possess a `fuel` method for all the vehicles. The `fuel` method has a JavaDoc-style comment and contains a `while` loop, which can be seen as syntactic sugar and is converted to a standard `for` loop during parsing. However, when later designing `Vehicle`'s subclasses, the user realises that bicycles cannot be fuelled and decides to do the push-down code refactoring, which removes the `fuel` method from `Vehicle` and pushes the method definition down to subclasses `Bus` and `Car` but not `Bicycle`. Instead of directly modifying the (program) `text`, most refactoring tools choose to parse the program text into its `ast`, perform code refactoring on the `ast`, and regenerate new (program) `text'`. The bottom-left corner of Figure 5.8

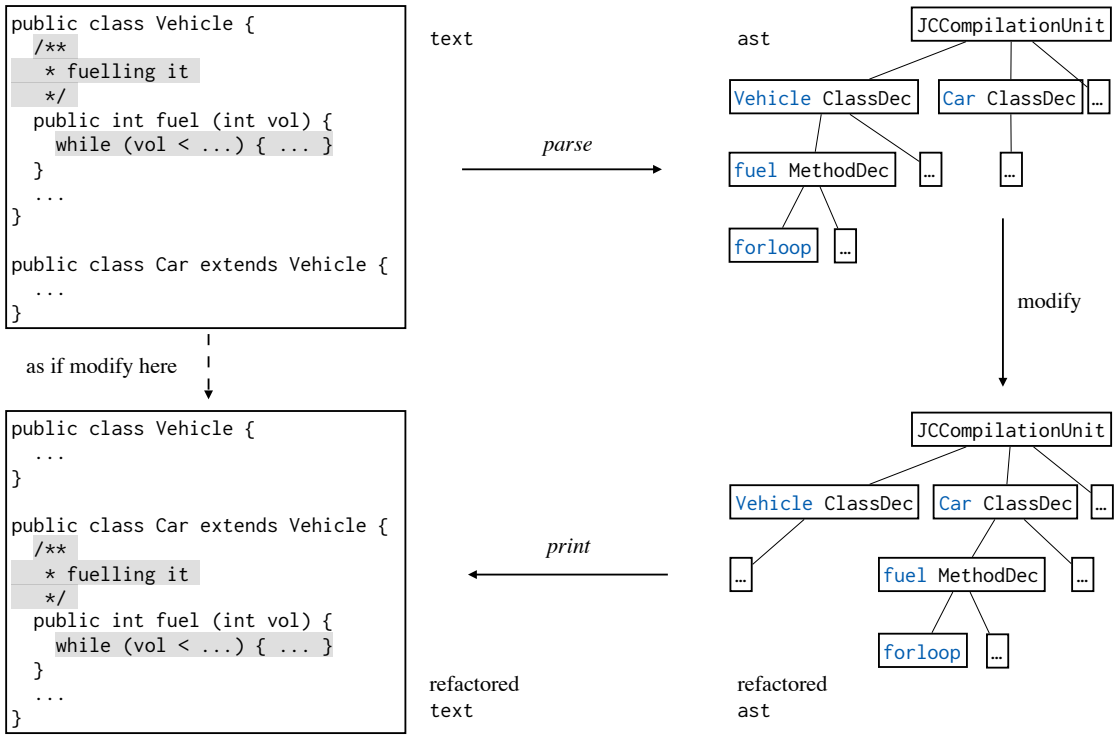


Figure 5.8: An example of the *push-down* code refactoring. (For simplicity, subclasses `Bus` and `Bicycle` are omitted.)

shows the desired (program) `text'` after refactoring, where we see that the comment associated with `fuel` is also pushed down, and the `while` sugar is kept. However, the preservation of the comment and syntactic sugar does not come for free actually, as the `ast`—being a concise and compact representation of the program `text`—includes neither comments nor the form of the original `while` loop. So if the user implements the *parse* and *print* functions as back-and-forth conversions between CSTs ASTs (or even as a well-behaved lens), they may produce unsatisfactory results in which the comment and the `while` syntactic sugar are lost.

5.4.1.2 Implementation in Our DSL

Implementation of the whole code refactoring tool for Java 8 using retentive lenses requires much engineering work. In this thesis, we focus on the theoretical foundation and language design, and have only implemented the transformation system (between CSTs and ASTs) for a small subset of Java 8 to demonstrate the possibility of having the whole system.

```
public class Vehicle {
  ...
}

public class Car extends Vehicle {
  public int fuel (int vol) {
    for (; vol < ... ;) {...}
  }
  ...
}
```

Figure 5.9: An unsatisfactory result after refactoring, losing the comment and *while* syntactic sugar.

Following the grammar of Java 8, we define data types for a simplified version of its concrete syntax, which consists of definitions of classes, methods, and variables; arithmetic expressions (including assignment and method invocation); and conditional and loop statements. For convenience, we also restrict the occurrence of statements and expressions to exactly once in some cases (such as variable declarations). Then we define the corresponding simplified version of the abstract syntax that follows the one defined by the JDT parser (Oracle Corporation and OpenJDK Community, 2014). This subset of Java 8 has around 80 CST constructs (production rules) and 30 AST constructs; the 70 consistency relations among them generate about 3000 lines of code for the retentive lenses and auxiliary functions (such as the ones for conversions between interchangeable data types and edit operations).

Now, we define the consistency relations. Since the structure of the consistency relations for the transformation system is roughly similar to the ones in Figure 5.4, here we only highlight two of them as examples; the reader can refer to the supplementary material to see the complete program. We see that for the concrete syntax everything is a class declaration (`ClassDecl`), while for the abstract syntax everything is a tree (`JCTree`). As a `ClassDecl` should correspond to a `JCClassDecl`, which by definition is yet not a `JCTree`, we use the constructors `FromJCStatement` and `FromJCClassDecl` to make it a `JCTree`, emulating the inheritance in Java. This is described by the consistency relation[1]

[1]We include keywords such as `class` in the CST patterns for improving readability, although they should be removed.

```
ClassDecl <---> JCTree
  NormalClassDeclaration0 _ "class" n "extends" sup body ~
  FromJCStatement (FromJCClassDecl (JCClassDecl N n (J (JCIdent sup)) body))

  NormalClassDeclaration1 _ mdf "class" n "extends" sup body ~
  FromJCStatement (FromJCClassDecl (JCClassDecl (J mdf) n (J (JCIdent sup)) body))
  ...  .
```

Depending on whether a class has a modifier (such as `public` and `private`) or not, the concrete syntax is divided into two cases while we use a `Maybe` type in the abstract syntax representing both cases. (To save space, the constructors `Just` and `Nothing` are shortened to `J` and `N` respectively.) Similarly, there are further two cases where a class does not extend some superclass and are omitted here.

Next, we see how to represent `while` loop using the basic `for` loop, as the abstract syntax of a language should be as concise as possible[1]:

```
Statement <---> JCStatement
  While "while" "(" exp ")" stmt ~ JCForLoop Nil exp Nil stmt
```

where the four arguments of `JCForLoop` in order denote (list of) initialisation statements, the loop condition, (list of) update expressions, and the loop body. As for a `while` loop, we only need to convert its loop condition `exp` and loop body `stmt` to AST types and put them in the correct places of the `for` loop. Initialisation statements and update expressions are left empty since there is none.

5.4.1.3 Demo

We can now perform some experiments on Figure 5.8.

- First we test `put cst ast ls`, where `(ast, ls) = get cst`. We get back the same `cst`, showing that the generated lenses do satisfy Hippocraticness.
- As a special case of Correctness, we let `cst' = put cst ast []` and check `fst (get cst') == ast`. In `cst'`, the `while` loop becomes a basic `for` loop and all the comments disappear. This shows that *put* will create a new source solely from the view if links are missing.
- Then we change `ast` to `ast'` and the set of links `ls` to `ls'` using our edit operations, simulating the *push-down* code refactoring for the `fuel` method. To show the effect of Retentiveness

[1] Although the JDT parser does not do this.

more clearly, when building `ast'`, the `fuel` method in the `Car` class is copied from the `Vehicle` class, while the `fuel` method in the `Bus` class is built from scratch (i.e. replaced with a 'new' `fuel` method). Let `cst' = put cst ast' ls'`. In the `fuel` method of the `Car` class, the `while` loop and its associated comments are preserved; but in the `fuel` method of the `Bus` class, there is only a `for` loop without any associated comments. This is where Retentiveness helps the user to retain information on demand. Finally, we also check that Correctness holds: `fst (get cst') == ast'`.

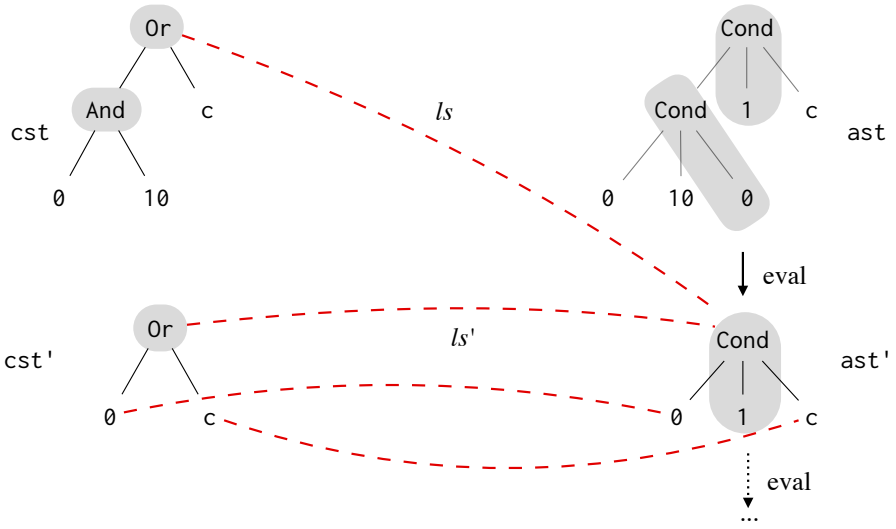
5.4.2 Resugaring Revisited

We have explained the notion of resugaring in Section 3.3.1 and used the TIGER language to illustrate how BIYACC helps to solve some simple cases of the resugaring problem. In this subsection, we demonstrate that the resugaring problem, rather than just simple cases, can be completely solved while leaving ASTs unmodified with the help of Retentiveness and our DSL: To retain syntactic sugar, we can write consistency relations between the surface syntax and the abstract syntax and pass the generated `put` function proper links.

We still take resugaring for TIGER as an example. As shown in Figure 4.4, we use names (i.e. data constructors) `Or` and `And` to represent *logical or* and *logical and* expressions in CSTs respectively; they will both be converted to *if-then-else* expressions represented by `TCond` (defined in Figure 3.6) in ASTs. Since the code for defining consistency relations for TIGER in this DSL is substantially similar to the code for defining parser and printer pairs in BIYACC, we only show the core part of the code regarding logical expressions.

```
Prmtv <---> Arith
  Or  l r  ~  Cond l (Num 1) r
  And l r  ~  Cond l r (Num 0)
  ...
```

With the generated `put`, resugaring for the one-step evaluated `ast' = Cond 0 1 c` from `ast = Cond (Cond 0 10 0) 1 c` (that is parsed from text `0 & 10 | c`) is illustrated in the following figure. For this simple case, we can obtain `ast'` and links `ls` by using the *replace* edit operation to replace `Cond 0 10 0` in `ast` with (the one-step evaluated result) `0`. As for `cst'`, the syntactic sugar `Or` is preserved, for Retentiveness requires $fst \cdot ls \subseteq fst \cdot ls'$.



Both Pombrio and Krishnamurthi's 'tag approach' and our 'link approach', in actuality, identifies where an AST construct comes from; however, the link approach has an advantage that it leaves ASTs clean and unmodified so that we do not need to patch up the existing compiler to deal with tags.

5.4.3 XML Synchronisation

In this subsection, we present a case study on XML synchronisation, which is pervasive in the real world and different from syntax tree manipulation. The specific example used in this subsection is from Pacheco et al.'s paper (Pacheco et al., 2014b), where they use their DSL, BiFluX, to synchronise address books.

As for their example, both the source address book and the view address book are grouped by social relationships; however, the source address book contains names, emails, and telephone numbers while the view (social) address book contains names only:

```
data AddrBook =
  AddrBook (List AddrGroup)
data AddrGroup =
  AddrGroup String (List Person)
data Person =
  Person (Triple Name Email Tel)
```

```
data SocialBook  =
  SocialBook (List SocialGroup)
data SocialGroup =
  SocialGroup String (List Name)
data Triple a b c= Triple a b c
type Name = String ...
```

To synchronise `AddrBook` and `SocialBook`, we write consistency relations in our DSL and the core ones are

```
AddrGroup  <---> SocialGroup
  AddrGroup grp p  ~  SocialGroup grp p

List Person <---> List Name
  Nil  ~  Nil
  Cons p xs  ~  Cons p xs
```

```
Person <---> Name
  Person t  ~  t

Triple Name Email Tel <---> Name
  Triple name  _ _  ~  name .
```

The consistency relations will compile to a pair of `get` and `put`.

As for their example, the original source `addrBook` and its consistent view `socialBook` are

```
AddrBook
  (Cons (AddrGroup "coworkers" (Cons
    (Person (Triple "Hugo Pacheco" "hugo@nii.ac.jp" "000111")) (Cons
    (Person (Triple "Zhenjiang Hu" "hu@nii.ac.jp" "222333")) Nil)))
  (Cons (AddrGroup "friends" (Cons
    (Person (Triple "John Doe" "doe@abc.xyz" "444555")) Nil)) Nil))
```

and

```
SocialBook
  (Cons (SocialGroup "coworkers" (Cons
    "Hugo Pacheco" (Cons
    "Zhenjiang Hu" Nil)))
  (Cons (SocialGroup "friends" (Cons
    "John Doe" Nil)) Nil))
```

respectively.

The view `socialBook` is updated in a way that we

1. reorder the two groups;
2. change Hugo's group (from `coworkers` to `friends`);
3. create a new social relationship group (`family`) for family members.

That is, if we assume that, in the social group `friends`, `Hugo Pacheco` is inserted after `John Doe`, the desired updated `socialBook'` should be

```
SocialBook
  (Cons (SocialGroup "friends" (Cons
    "John Doe" (Cons
    "Hugo Pacheco" Nil)))
  (Cons (SocialGroup "coworkers" (Cons
    "Zhenjiang Hu" Nil))
  (Cons (SocialGroup "family" Nil) Nil))) .
```

In our case, to produce `socialBook'`, we handle the three update steps using our basic edit operations (in this case, only *swap*, *move*, and *replace*) which also maintain the links. Feeding the original source `addrBook`, updated view `socialBook'` and links `hls'` to the (generated) `put` function, we obtain the following updated `addrBook'`:

```
AddrBook (Cons (AddrGroup "friends" (Cons
        (Person (Triple "John Doe" "doe@abc.xyz" "444555")) (Cons
        (Person (Triple "Hugo Pacheco" "hugo@nii.ac.jp" "000111")) Nil)))
  (Cons (AddrGroup "coworkers" (Cons
        (Person (Triple "Zhenjiang Hu" "hu@nii.ac.jp" "222333")) Nil))
  (Cons (AddrGroup "family" Nil) Nil))) .
```

It is clearly seen that carefully maintained links help us to preserve email addresses and telephone numbers associated with each person during the *put* process; note that the updated view is not consistent with the original source and in this case, well-behavedness does not guarantee information retention.

As pointed out by Pacheco et al., examples of this kind motivate extensions to alignment-aware languages such as Boomerang (Bohannon et al., 2008) and matching lenses (Barbosa et al., 2010). In fact, it is hard for those languages to handle source-view alignment of this kind, where some view elements are moved out of its original list-like structure (or *chunk* (Barbosa et al., 2010)) and put into a new list-like structure, probably far away—because when using those languages, we usually lift a lens combinator k handling a single element to $k*$ dealing with a list of elements, so that the 'scope' of the alignment performed by $k*$ is always set within that single list (it currently works on). Pacheco et al. overcome the problem by providing the functionality that allows us to write alignment strategies manually; in this way, when we see several lists at once, we are free to look up elements in all the lists[1].

[1]To be precise, in this specific social address book example, an element is still aligned within a single list only; however, when it becomes an 'unmatched' element, the user can write his own strategy such as reusing some elements in other lists for avoiding information loss, rather than creating a new one from scratch.

5.5 Related Work

5.5.1 Alignment

From the dawn of the bidirectional programming languages (Foster et al., 2007), *alignment* has been recognised as an important problem when we need to synchronise two lists—if a view element (in a list) is modified (e.g., inserted, deleted, or reordered), which source element should be matched with the (modified) view element and updated correspondingly? Our work on retentive lenses with links is closely related to this.

5.5.1.1 Alignment for Lists

The earliest lenses (Foster et al., 2007) only allow source and view elements to be matched positionally—the n-th source element is simply updated using the n-th element in the modified view. Later, lenses with more powerful matching strategies are proposed, such as dictionary lenses (Bohannon et al., 2008) and their successor matching lenses (Barbosa et al., 2010). In matching lenses, a source is divided into a *resource* consisting of *chunks* of information that can be reordered, and a 'rigid complement' storing information outside the chunk structure; the reorderable chunk structure is preserved in the view. When a *put* is invoked, it will first do source-view elements matching, which finds the correspondence between chunks of the old and new views using some predefined strategies; based on the correspondence, the resource is 'pre-aligned' to match the chunks in the new view. Then element-wise updates are performed on the aligned chunks. The design of matching lenses is to be practically easy to use, so they are equipped with a few fixed matching strategies (such as greedy align) from which the user can choose. However, whether the information is retained or not, still depends on the lens applied after matching. As a result, the more complex the applied lens is, the more difficult to reason about the information retained in the new source. (Moreover, they suffer a disadvantage that the alignment is only between a single source list and a single view list, as already discussed in the last paragraph of Section 5.4.3.)

BiFluX (Pacheco et al., 2014b) not only provides the user with align-by-position and align-by-key matching strategies as two primitives but also allows the user to write her/his own alignment strategies. In this way, when we see several lists at once, we are free to search for elements and match them in all the lists. But this alignment still has some limitations: (i) it works only for list-like data (data that can be flattened into a list); (ii) each source element and each view element can only be matched at most once—after that they are classified as either *matched pair*, *unmatched source element*, or *unmatched view element*. Assuming that an element in the view has been copied

several times, there is no way to align all the copies with the same source element. (However, it is possible to reuse an element several times for the handling of unmatched elements.)

By contrast, retentive lenses are designed to abstract out matching strategies (alignment) and are more like taking the result of matching as an additional input. This matching is not a one-layer matching but rather, a global one that produces (possibly all the) links between a source's and a view's unchanged parts. The information contained in the linked parts is preserved independently of any further applied lenses.

5.5.1.2 Alignment for Containers

To generalise list alignment, a more general notion of data structures called *containers* (Abbott et al., 2005) is used (Hofmann et al., 2012). In the container framework, a data structure is decomposed into a *shape* and its *content*; the shape encodes a set of positions, and the content is a mapping from those positions to the elements in the data structure. The existing approaches to container alignment take advantage of this decomposition and treat shapes and contents separately. For example, if the shape of a view container changes, Hofmann et al.'s approach will update the source shape by a fixed strategy that makes insertions or deletions at the rear positions of the (source) containers. By contrast, Pacheco et al.'s method permits more flexible shape changes, and they call it *shape alignment* (Pacheco et al., 2012). In our setting, both the consistency on data and the consistency on shapes are specified by the same set of consistency declarations. In the *put* direction, both the data and shape of a new source is determined by (computed from) the data and shape of a view, so there is no need to have separated data and shape alignments.

Container-based approaches have the same situation (as list alignment) that the retention of information is dependent on the basic lens applied after alignment. Besides, as a generalisation of list alignment, it is worth noting that separation of data alignment and shape alignment will hinder the handling of some algebraic data types. First, in practice, it is usually difficult for the user to define container data types and represent their data using containers. We use two mutually recursive data types `Expr` and `Term` defined in Figure 5.1 to illustrate. If the user wants to use containers to define them, one way might be to parametrise the types of terminals (leaves in a tree, here `Integer` only):

```
data Expr i  = Plus  (Expr i) (Term i)
             | Minus (Expr i) (Term i)
             | FromT (Term i)
data Term i  = Neg   (Term i)
             | Lit   i
             | Paren (Expr i)
data Arith i = Add   (Arith i) (Arith i)
             | Sub   (Arith i) (Arith i)
             | Num i
```

Here the terminals are of the same type `Integer`. However, imagine the situation where there are more than ten types of leaves, it is a boring task to parameterise all of them as type variables.

Moreover, the container-based approaches face another serious problem: they always translate a change on data in the view to another change on data in the source, without affecting the shape of a container. This is wrong in some cases, especially when the decomposition into shape and data is inadequate. For example, let the source be `Neg (Lit 100)` and the view `Sub (Num 0) (Num 100)`. If we modify the view by changing the integer `0` to `1` (so that the view becomes `Sub (Num 1) (Num 100)`), the container-based approach would not produce a correct source `Minus ...`, as this data change in the view must not result in a shape change in the source. In general, the essence of container-based approaches is the decomposition into shape and data such that they can be processed independently (at least to some extent), but when it comes to scenarios where such decomposition is unnatural (like the example above), container-based approaches can hardly help.

5.5.2 Provenance and Origin

Our idea of links is inspired by research on provenance (Cheney et al., 2009) in database communities and origin tracking (van Deursen et al., 1993) in the rewriting communities.

Cheney et al. classify provenance into three kinds, *why*, *how*, and *where*: *why-provenance* is the information about which data in the view is from which rows in the source; *how-provenance* additionally counts the number of times a row is used (in the source); *where-provenance* in addition records the column where a piece of data is from. In our setting, we require that two pieces of data linked by vertical correspondence be equal (under a specific pattern), and hence the vertical correspondence resembles where-provenance. Leaping from database communities to programming language communities, we find that the above-mentioned provenance is not powerful enough as they are mostly restricted to relational data, namely rows of tuples. In functional programming, the algebraic data types are more complex (represented as sums of

products), and a view is produced by more general functions rather than relational operations such as selection, projection, and join. For this need, *dependency provenance* (Cheney et al., 2011) is proposed; it tells the user on which parts of a source the computation of a part of a view depends. In this sense, our consistency links are closer to dependency provenance.

The idea of inferring consistency links can be found in the work on origin tracking for term rewriting systems (van Deursen et al., 1993), in which the origin relations between rewritten terms can be calculated by analysing the rewrite rules statically. However, it was developed solely for building traces between intermediate terms rather than using trace information to update a tree further. Based on origin tracking, de Jonge and Visser implemented an algorithm for code refactoring systems, which 'preserves formatting for terms that are not changed in the (AST) transformation, although they may have changes in their subterms' (de Jonge and Visser, 2012). This description shows that the algorithm also decomposes large terms into smaller ones resembling our regions. Therefore, in terms of the formatting aspect, we think that retentiveness can be in effect the same as their theorem if we adopt the 'square diagram' (see Section 5.6.2). However, they only tailored the theorem for their specific printing algorithm but did not generalise the theorem to other scenarios.

Similarly, Martins et al. developed a system for attribute grammars which define transformations between tree structures (in particular CSTs and ASTs) (Martins et al., 2014). Their bidirectional system also uses links to trace the correspondence between source nodes and view nodes, which is later used by *put* to solve the syntactic sugar problem. The differences between their system and ours are twofold: One is that in their system, links are implicitly used in the *put* direction. The advantage (of implicit link usage) is that, for the user, links become transparent and are automatically maintained when a view is updated; the disadvantage is that, as a result, newly created nodes on an AST can never have 'links back' to the old CST, even if they might be the copies of some old nodes. The second difference is the granularity of links; in their system, a link seems to connect the whole subtrees between a CST and an AST instead of between smaller regions. As a result, if a leaf of an AST is modified, all the nodes belonging to the spine from the leaf to the root will lose their links.

The use of consistency links can also be found in Wang et al.'s work, where the authors extend state-based lenses and use links for tracing data in a view to its origin in a source (Wang et al., 2011). When a sub-term in the view is edited locally, they use links to identify a sub-term in the source that 'contains' the edited sub-term in the view. When updating the old source, it is sufficient to only perform state-based *put* on the identified sub-term (in the source) so that the update becomes an incremental one. Since lenses generated by our DSL also create consistency links (albeit for a

different purpose), they can be naturally incrementalised using the same technique.

5.5.3 Operation-based BX

Our work is closely relevant to the operation-based approaches to BX, in particular, the delta-based BX model (Diskin et al., 2011a,b) and edit lenses (Hofmann et al., 2012). The (asymmetric) delta-based BX model regards the differences between a view state v and v' as *deltas*, which are abstractly represented as arrows (from the old view to the new view). The main law of the framework can be described as 'given a source state s and a view delta det_v, det_v should be translated to a source delta det_s between s and s' satisfying $get\ s' = v'$'. As the law only guarantees the existence of a source delta det_s that updates the old source to a correct state, it is yet not sufficient to derive Retentiveness in their model, for there are infinite numbers of translated delta det_s which can take the old source to a correct state, of which only a few are 'retentive'. To illustrate, Diskin et al. tend to represent deltas as edit operations such as *create*, *delete*, and *change*; representing deltas in this way will only tell the user what must be changed in the new source, while it requires additional work to reason about what is retained. However, it is possible to exhibit Retentiveness if we represent deltas in some other proper form. Compared to Diskin et al.'s work, Hofmann et al. give concrete definitions and implementations for propagating edit operations (in a symmetric setting).

5.6 Discussions

In this chapter, we showed that well-behavedness is not sufficient for retaining information after an update and it may cause problems in many real-world applications such as code refactoring. To address the issue, we illustrated how to use links to preserve desired data fragments of the original source, and developed a semantic framework of (asymmetric) retentive lenses for region models. Then we presented a small DSL tailored for describing consistency relations between syntax trees; we showed its syntax, semantics, and proved that the pair of *get* and *put* functions generated from any program in the DSL form a retentive lens. We further illustrated the practical use of retentive lenses by giving examples in the field of code refactoring, XML synchronisation, and resugaring. In the related work, we discussed the relations with alignment, origin tracking, and operation-based BX. At the end of the chapter, we will briefly discuss the integration of Retentiveness into BiYacc, Strong Retentiveness (that subsumes Hippocraticness), our thought on (retentive) lens composition, the feasibility of retaining code styles for refactoring tools, and our

choice of the word ‘retentive’.

5.6.1 BiYacc and Retentive Printing

BiYacc does not support ‘the real retentive printing (formalised in this chapter). Why do not we design BiYacc in a way that supports it from the very beginning? The truth is that, two years after BiYacc’s birth, we realised that state-based lenses, even being put-based, are not powerful enough for our application scenario, so that we have to devise a more powerful lens framework, and that gives birth to retentive lenses and the small experimental DSL introduced in this chapter.

The DSL (introduced in this chapter) itself is closely related to but simpler than BiYacc, and retentive lenses generated from the DSL can theoretically be used as a new back end of BiYacc and to achieve *retentive printing*. The idea is appealing, yet we have not integrated retentive lenses into BiYacc because of the conflicts of their design philosophies: as mentioned very early in this thesis before, BiYacc is state-based and put-based, while retentive lenses are (champion) neither of them. Retentive lenses seem to be a framework that lies in between: It is not state-based because, in addition to the states of a pair of source and view, the *put* function also accepts a set of links recording trace information. It is not delta-based or operation-based because the set of links are between a source and a view instead of a view and its modified one; as a result, the *put* function does not produce edits/delta/correspondence between an old source and its updated one, either. It is not put-based since the user explicitly specifies consistency relations between sources and views rather than implementing *put* functions (which implicitly derive consistency relations). Finally, it is not get-based since the user needs to supply links as an additional input to *put*—although it can be left empty—which means that the user still sometimes needs to consider the *put* behaviour.

We leave the integration of retentive lenses into BiYacc-like tools as future work. Nonetheless, it should not be too difficult, even considering bi-filters—which, fortunately, have been designed in a modular way and should not hurt retentiveness—but the real problems are, for example, (i) how to create consistency links for (the consistency relations extracted from) printing strategies which involve more complex features such as deep patterns and (ii) how to let the user easily obtain the links (supplied to *put*) (s)he wants.

5.6.2 Opting for Triangular Diagrams

The reader may wonder why do not we formalise Retentiveness and retentive lenses in a way that it satisfies a square commutative diagram

$$hls \cdot vls = vls' \cdot hls'$$

where

$$\begin{aligned}(v, hls) &= get\ s \\ (v', vls) &= modify\ v \text{ (for some view-modifying function } modify) \\ (s', vls') &= put\ (s, v', vls) \\ (v', hls') &= get\ s'\ .\end{aligned}$$

That is, *put* takes some *'vertical links'* between a view and its modified version and produces vertical links between an old source and its updated version, and the composition of the old consistency links *hls* and vertical links *vls* is equal to the composition of the new vertical links vls' and consistency links hls'. A possible design for vertical links is to represent a link as two paths sharing the same region pattern, i.e. ($vpath$, $vpat$, $vpath'$)—meaning that a data fragment ($vpat$, $vpath$) in a view is not destroyed but probably moved to $vpath'$ in the updated view. When composing horizontal links with vertical links, we first transform each vertical link ($vpath$, $vpat$, $vpath'$) to an equivalent representation (($vpat$, $vpath$), ($vpat$, $vpath'$)) and then reuse the link composition introduced in Section 5.1.5.

Including such vertical links representing view updates in the theory was something we thought about (for quite some time), but eventually, we opted for the current, simpler theory. The rationale is that the original state-based lens framework (which we extended) does not really have the notion of view update built in. A view given to a *put* function is not necessarily modified from the view got from the source—it can be constructed in any way, and *put* does not have to consider how the view is constructed. Coincidentally, the paper about (symmetric) delta-based lenses (Diskin et al., 2011b) also introduces square diagrams and later switches to triangular diagrams. We retain this separation of concern in our framework, in particular separating the jobs of retentive lenses and third-party tools.

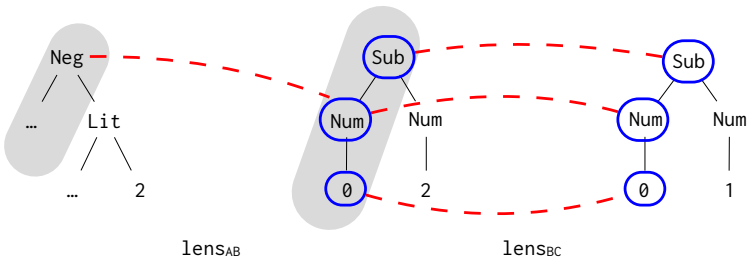
5.6.3 Strong Retentiveness

Through our research into Retentiveness, we also tried a different theory, which we call *Strong Retentiveness* now, that requires that the consistency links generated by *get* should capture all the 'information' of the source and *uniquely identify* it. Strong Retentiveness is appealing in the sense that (we proved that) it subsumes Hippocraticness: the more information we require that the new source have, the more restrictions we impose on the possible forms of the new source; in the extreme case where the input links are consistency links—which capture all the information and can only be satisfied by at most one source—the new source has to be the same as the original one. However, using Strong Retentiveness demands extra effort in practice, for a set of region patterns can never uniquely identify a tree; as a result, much more information is required. For instance, $\texttt{cst}_1$ `= Minus "" (Lit 1) (Lit 2)` has region patterns $\texttt{reg}_1$ `= Minus "" _ _`, $\texttt{reg}_2$ `= Lit 1`, and $\texttt{reg}_3$ `= Lit 2`, which, however, are also satisfied by $\texttt{cst}_2$ `= Minus "" (Lit 2) (Lit 1)` in which regions are assembled in a different way.

This observation inspires us to generalise region patterns to *properties* in order for holding more information (that can eventually uniquely identify a tree) and generalise links connecting *Pattern* × *Path* to links connecting *Property* × *Path* accordingly. We eventually formalised three kinds of properties that are sufficient to capture all the information of a tree (e.g. $\texttt{cst}_1$): *region patterns* (e.g. $\texttt{reg}_1$, $\texttt{reg}_2$, and $\texttt{reg}_3$), *relative positions* between two regions (e.g. $\texttt{reg}_2$ is the first child of $\texttt{reg}_1$ and $\texttt{reg}_3$ is the second child of $\texttt{reg}_1$), and *top* that marks the top of a tree (e.g. $\texttt{reg}_1$ is the top). Worse still, observant readers might have found that properties need to be named so that they can be referred to by other properties; for instance, the region pattern `Minus "" _ _` is named $\texttt{reg}_1$ and is referred to as the top of $\texttt{cst}_1$. This will additionally cause many difficulties in lens composition, as different lenses might assign the same region different names and we need to do 'alpha conversion'. Take everything into consideration, finally, we opted for the 'weaker' but simpler version of Retentiveness.

5.6.4 Rethinking Lens Composition

We defined retentive lens composition (Definition 5.1.9) in which we treat link composition as relation composition. In this case, however, the composition of two lenses $lens_{AB}$ and $lens_{BC}$ may not be satisfactory because the link composition might (trivially) produce an empty set as the result, if $lens_{AB}$ and $lens_{BC}$ decompose a tree b (of type B) in a different way, as the following example shows:



In the above figure, $\texttt{lens}_{\texttt{AB}}$ connects the region (pattern) `Neg a _` with `Sub (Num 0) _` (the grey parts); while $\texttt{lens}_{\texttt{BC}}$ decomposes `Sub (Num 0) _` into three small regions and establishes links for them respectively. For this case, our current link composition simply produces an empty set as the result.

Coincidentally, similar problems can also be found in quotient lenses (Foster et al., 2008): A quotient lens operates on sources and views that are divided into many equivalent classes, and the well-behavedness is defined on those equivalent classes rather than a particular pair of source and view. In order to establish a sequential composition $l; k$, the authors require that the abstract (view-side) equivalence relation of lens l is identical to the concrete (source-side) equivalence of lens k. We leave other possibilities of link composition to future work.

As for our DSL, the lack of composition does not cause problems because of the philosophy of design. Take the scenario of writing a parser for example where there are two main approaches for the user to choose: to use parser combinators (such as Parsec) or to use parser generators (such as Happy). While parser combinators offer the user many small composable components, parser generators usually provide the user with a high-level syntax for describing the grammar of a language using production rules (associated with semantic actions). Then the generated parser is used as a 'standalone black box' and usually will not be composed with some others (although it is still possible to be composed 'externally'). Our DSL is designed to be a 'lens generator' and we have no difficulty in writing bidirectional transformations for the subset of Java 8 in Section 5.4.1.

5.6.5 Retaining Code Styles

A challenge to refactoring tools is to retain the style of program text such as indentation, vertical alignment of identifiers, and the place of line breaks. For example, an argument of a function application may be vertically aligned with a previous argument; when a refactoring tool moves the application to a different place, what should be retained is not the absolute number of spaces preceding the arguments but the *property* that these two arguments are vertically aligned.

Although not implemented in the DSL, these properties can be added to the set of *Property* as introduced in Section 5.6.3.For instance, we may have VertAligned $x\ y \in Property$ for $x, y \in Name$ (i.e. x and y are names of some regions); a CST satisfies such a property if region y is vertically aligned with region x.When *get* computes an AST from such a vertically aligned argument and produces consistency links, the links will not include (real) spaces preceding the argument as a part of the source region; instead, the links connect the property VertAligned $x\ y$ (and the corresponding AST region).In the *put* direction, such links serve as directives to adjust the number of spaces preceding the argument to conform to the styling rule. In general, handling code styles can be very language-specific and is beyond the scope of this thesis but could be considered a direction of future work.

5.6.6 The Word Retentive

We are not the first to use the word retentive; the use of the word appeared in the paper on symmetric lenses (Hofmann et al., 2011), which is the pioneering work of edit lenses (Hofmann et al., 2012). In the paper (Hofmann et al., 2011), they defined both retentive sum lenses and forgetful sum lenses, where the 'retentive one keeps complements for both sub-lenses (branches) while the forgetful one keeps only one complement corresponding to the last put branch.' The usage is the same there: retentive is used to describe the kind of sum lenses that are able to memory and preserve more information.

6 Conclusions

We showed the necessity of a retentive printer, which should reconcile the compactness of ASTs and printing quality. We solved the problem by proposing a DSL, BiYacc, whose program denotes a pair of well-behaved parser and retentive printer for an unambiguous CFG; the parser and retentive printer are, in actuality, formed by the composition of an isomorphism between program text and CSTs, and a lens between CSTs and ASTs. We show that BiYacc facilitates many tasks such as resugaring, language evolution, and simple refactoring by demonstrating them in a medium-size case study on the Tiger language. After that, we improved our solution in two directions: one for the isomorphism part and the other for the lens part. For the isomorphism part, we supported (fully-disambiguated) ambiguous grammars by generalised parsing and bi-filters; in the parsing direction, bi-filters remove incorrect CSTs produced from some generalised parser; in the printing direction, bi-filters repair a potentially incorrect CST produced from some *put* function (for synchronising CSTs and ASTs). Bi-filters are carefully bidirectionalised so that they will not break the well-behavedness laws—provided that they satisfy our proposed bidirectioanlised filter laws. We extend BiYacc with this functionality and design accessible directives for specifying production rules' associativity and (relative) priorities, which give rise to compositional and commutative filters; power users can also define their own filters. We demonstrate the extended BiYacc using

the same Tiger language defined by an ambiguous grammar and a set of disambiguation rules, which also includes a manually written one for the dangling else problem. For the lens part, by putting forward retentive lenses and Retentiveness, we guarantee precise information retention such as comments and syntactic sugar in the situation where the input source (i.e. CSTs) and view (i.e. ASTs) are not consistent. We verify our idea by introducing a new DSL for writing tree transformations (especially between CSTs and ASTs) and presenting case studies on code refactoring, resugaring, and XML synchronisation. Retentive lenses are incompatible with either state-based lenses or put-based design, and we leave the integration of retentive lenses into BiYacc or developing some other new DSL supporting (true) retentive printing as possible future work. Related work and other future work are discussed in each individual chapters.

Bibliography

Annika Aasa. 1995. Precedences in Specifications and Implementations of Programming Languages. In *Selected Papers of the Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 3–26. http://dl.acm.org/citation.cfm?id=203429.203431

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (Sept. 2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2018. Introduction to Bidirectional Transformations. In *Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*, Jeremy Gibbons and Perdita Stevens (Eds.). Springer International Publishing, Cham, 1–28. https://doi.org/10.1007/978-3-319-79108-1_1

Ali Afroozeh and Anastasia Izmaylova. 2015. Faster, Practical GLL Parsing. In *Compiler Construction*, Björn Franke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–108. https://doi.org/10.1007/978-3-662-46663-6_5

A. V. Aho, S. C. Johnson, and J. D. Ullman. 1975. Deterministic Parsing of Ambiguous Grammars. *Commun. ACM* 18, 8 (Aug. 1975), 441–452. https://doi.org/10.1145/360933.360969

Andrew W. Appel. 1998. *Modern Compiler Implementation in ML, First Edition*. Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/CBO9780511811449

François Bancilhon and Nicolas Spyratos. 1981. Update Semantics of Relational Views. *ACM Transactions on Database Systems* 6, 4 (Dec. 1981), 557–575. http://doi.acm.org/10.1145/319628.319634

Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. 2010. Matching Lenses: Alignment and View Update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1863543.1863572

Richard Bird. 2014. *Thinking Functionally with Haskell*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9781316092415

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 407–419. https://doi.org/10.1145/1328438.1328487

Richard J. Boulton. 1996. *Syn: a single language for specifiying abstract syntax tress, lexical analysis, parsing and pretty-printing.* Technical Report UCAM-CL-TR-390. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-390.ps.gz

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. 2008. Dual Syntax for XML Languages. *Information Systems* 33, 4-5 (June 2008), 385–406. https://doi.org/10.1016/j.is.2008.01.006

David G. Cantor. 1962. On The Ambiguity Problem of Backus Systems. *J. ACM* 9, 4 (Oct. 1962), 477–479. https://doi.org/10.1145/321138.321145

James Cheney, Amal Ahmed, and Umut a. Acar. 2011. Provenance As Dependency Analysis. *Mathematical Structures in Computer Science* 21, 6 (Dec. 2011), 1301–1337. https://doi.org/10.1017/S0960129511000211

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (April 2009), 379–474. https://doi.org/10.1561/1900000006

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT '09)*. Springer-Verlag, Berlin, Heidelberg, 260–283. https://doi.org/10.1007/978-3-642-02408-5_19

Umeshwar Dayal and Philip A. Bernstein. 1982. On the Correct Translation of Update Operations on Relational Views. *ACM Trans. Database Syst.* 7, 3 (Sept. 1982), 381–416. https://doi.org/10.1145/319732.319740

Maartje de Jonge and Eelco Visser. 2012. An Algorithm for Layout Preservation in Refactoring Transformations. In *Software Language Engineering*, Anthony Sloane and Uwe Aßmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–59. https://doi.org/10.1007/978-3-642-28830-2_3

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011a. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10 (2011), 6:1–25. http://www.jot.fm/contents/issue_2011_01/article6.html

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011b. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *Model*

Driven Engineering Languages and Systems, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–318. https://doi.org/10.1007/978-3-642-24485-8_22

Jonas Duregård and Patrik Jansson. 2011. Embedded Parser Generators. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 107–117. https://doi.org/10.1145/2034675.2034689

Jay Earley. 1970. An Efficient Context-free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. 2015. The Essence of Bidirectional Programming. *Science China Information Sciences* 58, 5 (May 2015), 1–21. https://doi.org/10.1007/s11432-015-5316-8

John Nathan Foster. 2009. *Bidirectional Programming Languages*. Ph.D. Dissertation. University of Pennsylvania. https://repository.upenn.edu/edissertations/56/

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Language Systems* 29, 3, Article 17 (May 2007). https://doi.org/10.1145/1232420.1232424

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. *SIGPLAN Not.* 43, 9 (Sept. 2008), 383–396. https://doi.org/10.1145/1411203.1411257

Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA. https://www.csie.ntu.edu.tw/~r95004/Refactoring_improving_the_design_of_existing_code.pdf

Peter Friese, Sven Efftinge, and Jan Köhnlein. 2008. Xtext - Language Engineering for Everyone! https://www.eclipse.org/Xtext/

James Gosling, Bill Joy, and Guy Steele. 2006. The Java Language Specification, Third Edition. https://docs.oracle.com/javase/specs/

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. The Java Language Specification, Java SE 8 Edition (Java Series). https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Georg Gottlob, Paolo Paolini, and Roberto Zicari. 1988. Properties and Update Semantics of Consistent Views. *ACM Trans. Database Syst.* 13, 4 (Oct. 1988), 486–524. https://doi.org/10.1145/49346.50068

John Hennessy. 1982. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Language Systems* 4, 3 (July 1982), 323–344. http://doi.acm.org/10.1145/357172.357173

Martin Hirzel and Kristoffer Høgsbro Rose. 2013. Tiger Language Specification. https://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001/tiger-spec.pdf

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric Lenses. *SIGPLAN Not.* 46, 1 (Jan. 2011), 371–384. https://doi.org/10.1145/1925844.1926428

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 495–508. https://doi.org/10.1145/2103656.2103715

Zhenjiang Hu and Hsiang-Shang Ko. 2018. Principles and Practice of Bidirectional Programming in BiGUL. In *Bidirectional Transformations: International Summer School, Oxford, UK, July 25–29, 2016, Tutorial Lectures*, Jeremy Gibbons and Perdita Stevens (Eds.). Springer International Publishing, Cham, 100–150. https://doi.org/10.1007/978-3-319-79108-1_4

Michael Johnson, Robert D Rosebrugh, et al. 2016. Unifying Set-Based, Delta-Based and Edit-Based Lenses. In *Proceedings of the 5th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software (ETAPS 2016)*, Anthony Anjorin and Jeremy Gibbons (Eds.), Vol. 1571. CEUR Workshop Proceedings, Eindhoven, The Netherlands, 1–13. http://ceur-ws.org/Vol-1571/paper_13.pdf

Stephen C. Johnson. 1975. Yacc: Yet Another Compiler-Compiler. AT&T Bell Laboratories Technical Reports (AT&T Bell Laboratories Murray Hill, New Jersey 07974) (32). http://dinosaur.compilertools.net/yacc/

Brian W. Kernighan and Dennis M. Ritchie. 1989. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA. https://dl.acm.org/doi/book/10.5555/100511

Daisuke Kinoshita and Keisuke Nakano. 2017. Bidirectional Certified Programming. In *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software (ETAPS 2017)*, Romina Eramo and Michael Johnson (Eds.), Vol. 1827. CEUR Workshop Proceedings, Uppsala, Sweden, 31–38. http://ceur-ws.org/Vol-1827/paper7.pdf

Paul Klint and Eelco Visser. 1994. Using Filters for the Disambiguation of Context-Free Grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, G. Pighizzini and P. San Pietro (Eds.). University of Milan, Italy, Milano, Italy, 1–20. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.9812

Hsiang-Shang Ko and Zhenjiang Hu. 2018. An Axiomatic Basis for Bidirectional Programming. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 41 (Dec. 2018), 29 pages. https://doi.org/10.1145/3158129

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial*

Evaluation and Program Manipulation (PEPM '16). ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2847538.2847544

Wilf R. LaLonde and Jim des Rivieres. 1981. Handling Operator Precedence in Arithmetic Expressions with Tree Transformations. *ACM Transactions on Programming Language Systems* 3, 1 (Jan. 1981), 83–103. https://doi.org/10.1145/357121.357127

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179

David Lutterkort. 2008. Augeas—A configuration API. In *Proceedings of the Ottawa Linux Symposium*. Ottawa, Canada, 47–56. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.607.5792

Nuno Macedo, Hugo Pacheco, Alcino Cunha, and José Nuno Oliveira. 2013. Composing least-change lenses. *Proceedings of the Second International Workshop on Bidirectional Transformations* 57 (2013), 1–19. https://doi.org/10.14279/tuj.eceasst.57.868

Simon Marlow et al. 2010. Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/

Simon Marlow and Andy Gill. 2001. The Parser Generator for Haskell. https://www.haskell.org/happy/

Pedro Martins, João Saraiva, João Paulo Fernandes, and Eric Van Wyk. 2014. Generating Attribute Grammar-based Bidirectional Transformations from Rewrite Rules. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 63–70. https://doi.org/10.1145/2543728.2543745

Kazutaka Matsuda, Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2010. A Grammar-Based Approach to Invertible Programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*, Andrew D. Gordon (Ed.). Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 448–467. https://doi.org/10.1007/978-3-642-11957-6_24

Kazutaka Matsuda and Meng Wang. 2018a. Embedding Invertible Languages with Binders: A Case of the FliPpr Language. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 158–171. http://doi.org/10.1145/3242744.3242758

Kazutaka Matsuda and Meng Wang. 2018b. FliPpr: A System for Deriving Parsers from Pretty-Printers. *New Generation Computing* 36, 3 (01 Jul 2018), 173–202. https://doi.org/10.1007/s00354-018-0033-7

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Dissertation. Chalmers University of Technology. http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

Oracle Corporation and OpenJDK Community. 2014. OpenJDK. http://openjdk.java.net/

Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. 2012. Delta lenses over inductive types. *Electronic Communications of the EASST* 49 (2012), 1–17. https://doi.org/10.14279/tuj.eceasst.49.713

Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014a. Monadic Combinators for "Putback" Style Bidirectional Programming. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2543728.2543737

Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014b. BiFluX: A Bidirectional Functional Update Language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP '14)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/2643135.2643141

Hugo José Pereira Pacheco. 2012. *Bidirectional Data Transformation by Calculation.* Ph.D. Dissertation. University of Minho. https://www.di.uminho.pt/~hpacheco/publications/phdthesis.pdf

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 361–371. https://doi.org/10.1145/2594291.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 75–87. https://doi.org/10.1145/2784731.2784755

Tillmann Rendel and Klaus Ostermann. 2010. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1863523.1863525

Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (Sept. 2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: A cubic Tomita-style GLR parsing algorithm. *Acta Informatica* 44, 6 (2007), 427–461. https://doi.org/10.1007/s00236-007-0054-z

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Perdita Stevens. 2008. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling* 9, 1 (Dec. 2008), 7. https://doi.org/10.1007/s10270-008-0109-9

Masaru Tomita. 1985. An Efficient Context-free Parsing Algorithm for Natural Languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 756–764. http://dl.acm.org/citation.cfm?id=1623611.1623625

V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010, Article 3 (Jan. 2010), 26 pages. https://doi.org/10.1155/2010/602570

Mark van den Brand and Eelco Visser. 1996. Generation of Formatters for Context-free Languages. *ACM Transactions on Software Engineering and Methodology* 5, 1 (Jan. 1996), 1–41. https://doi.org/10.1145/226155.226156

Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, London, UK, 365–370. https://doi.org/10.1007/3-540-45306-7_26

Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, 143–158. https://doi.org/10.1007/3-540-45937-5_12

A. van Deursen, P. Klint, and F. Tip. 1993. Origin Tracking. *Journal of Symbolic Computation* 15, 5-6 (May 1993), 523–545. https://doi.org/10.1016/S0747-7171(06)80004-0

Eelco Visser. 1997a. A Case Study in Optimizing Parsing Schemata by Disambiguation Filters. In *International Workshop on Parsing Technology (IWPT 1997)*. Massachusetts Institute of Technology, Boston, USA, 210–224. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.1339

Eelco Visser. 1997b. *Syntax Definition for Language Prototyping*. Ph.D. Dissertation. University of Amsterdam. https://dare.uva.nl/search?identifier=4a30326d-626f-4355-a2e7-29b0239e975f

Meng Wang, Jeremy Gibbons, and Nicolas Wu. 2011. Incremental Updates for Efficient Bidirectional Transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/2034773.2034825

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 10, 2 (1967), 189 – 208. http://www.sciencedirect.com/science/article/pii/S001999586780007X

Zirun Zhu, Hsiang-Shang Ko, Pedro Miguel Ribeiro Martins, João Alexandre Saraiva, and Zhenjiang Hu. 2015. BiYacc: Roll Your Parser and Reflective Printer into One. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations (STAF 2015)*, Alcino Cunha and Ekkart Kindler (Eds.), Vol. 1396. CEUR Workshop Proceedings, L'Aquila, Italy, 43–50. http://ceur-ws.org/Vol-1396/p43-zhu.pdf

Zirun Zhu, Hsiang-Shang Ko, Yongzhe Zhang, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2020a. Parsing and Reflective Printing, Bidirectionally. *New Generation Computing* (2020), to appear. https://doi.org/10.1007/s00354-019-00082-y

Zirun Zhu, Zhixuan Yang, Hsiang-Shang Ko, and Zhenjiang Hu. 2020b. Retentive Lenses. (2020). https://arxiv.org/abs/2001.02031

Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2016. Parsing and Reflective Printing, Bidirectionally. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/2997364.2997369

A

Proofs about Retentive Lenses

A.1 Composability

In this section, we show the proof of Theorem 5.1.10 with the help of Figure 5.3 and the definition of retentive lens composition (Definition 5.1.9).

Hippocraticness Preservation. We prove that the composite lens satisfies Hippocraticness with the help of Figure 5.3 and the definition of retentive lens composition (Definition 5.1.9).
Let $get_{AC}\ a = (c, ls_{ac})$. We prove $put_{AC}\ (a, c', ls_{ac'}) = a' = a$. In this case, $c' = c$ and $ls_{ac'} = ls_{ac}$.

$$
\begin{aligned}
&put_{AC}\ (a, c', ls_{ac'}) \\
=&\{\ put_{AC}\ (a, c', ls_{ac'}) = a' = put_{AB}\ (a, b', ls_{ab'})\ \} \\
&put_{AB}\ (a, b', ls_{ab'}) \\
=&\{\ ls_{ab'} = ls_{ac'} \cdot ls^{\circ}_{b'c'}\ \} \\
&put_{AB}\ (a, b', ls_{ac'} \cdot ls^{\circ}_{b'c'}) \\
=&\{\ \text{Since } c' = c, \text{ we have } ls_{ac'} = ls_{ac} \text{ and } ls_{b'c'} = ls_{b'c}\ \} \\
&put_{AB}\ (a, b', (ls_{ac} \cdot ls^{\circ}_{b'c})^{\circ})
\end{aligned}
$$

$$
\begin{aligned}
&=\{\ b' = put_{BC}\ (b, c', ls_{bc'})\ \text{and}\ c' = c\ \} \\
&\quad put_{AB}\ (a, put_{BC}\ (b, c, ls_{bc}), ls_{ac} \cdot ls^{\circ}_{b'c}) \\
&=\{\ \text{By Hippocraticness of } lens_{BC},\ b' = put_{BC}\ (b, c, ls_{bc}) = b\ \} \\
&\quad put_{AB}\ (a, b, ls_{ac} \cdot ls^{\circ}_{bc}) \\
&=\{\ \text{The link composition is ⑥ in Figure 5.3, and } b' = b\ \} \\
&\quad put_{AB}\ (a, b, ls_{ab}) \\
&=\{\ \text{Hippocraticness of } lens_{AB}\ \} \\
&\quad a\ .
\end{aligned}
$$

□

Correctness Preservation. We prove that the composite lens satisfies Correctness with the help of Figure 5.3 and the definition of retentive lens composition (Definition 5.1.9).
Let $a' = put_{AC}\ (a, c', ls_{ac'})$, we prove $fst\ (get_{AC}\ a') = c'$.

$$
\begin{aligned}
&\quad fst\ (get_{AC}\ a') \\
&=\{\ \text{Definition of } get_{AC}\ \} \\
&\quad fst\ (get_{BC}\ (fst\ (get_{AB}\ a'))) \\
&=\{\ a' = put_{AC}\ (a, c', ls_{ac'})\ \} \\
&\quad fst\ (get_{BC}\ (fst\ (get_{AB}\ (put_{AC}\ (a, c', ls_{ac'}))))) \\
&=\{\ put_{AC}\ (a, c', ls_{ac'}) = a' = put_{AB}\ (a, b', ls_{ab'})\ \} \\
&\quad fst\ (get_{BC}\ (fst\ (get_{AB}\ (put_{AB}\ (a, b', ls_{ab'}))))) \\
&=\{\ \text{Correctness of } lens_{AB}\ \} \\
&\quad fst\ (get_{BC}\ (b')) \\
&=\{\ b' = put_{BC}\ (b, c', ls_{bc'})\ \} \\
&\quad fst\ (get_{BC}\ (put_{BC}\ (b, c', ls_{bc'}))) \\
&=\{\ \text{Correctness of } lens_{BC}\ \} \\
&\quad c'\ .
\end{aligned}
$$

□

Retentiveness Preservation. In Figure 5.3, we prove $fst \cdot ls_{ac} \subseteq fst \cdot ls_{a'c'}$.

To finish the proof, we need the following lemma.

Lemma A.1.1. Given a relation R and a function f, we have

$$\textsc{rdom}\,(f \cdot R) = \textsc{rdom}\,R \quad \text{if } \textsc{ldom}\,R \subseteq \textsc{rdom}\,f\text{, and}$$
$$\textsc{ldom}\,(R \cdot f) = \textsc{ldom}\,R \quad \text{if } \textsc{rdom}\,R \subseteq \textsc{ldom}\,f\text{.}$$

Proof. We prove the first equation; the second equation is symmetric.
Suppose $f : X \to Y$ and $R : Y \sim Z$. By definition, $\textsc{rdom}\,R = \{\, z \in Z \mid \exists y \in Y,\ y\ R\ z \,\}$ and $\textsc{rdom}\,(f \cdot R) = \{\, z \in Z \mid \exists y \in Y,\ \exists\, x \in X,\ x\ f\ y\ R\ z \,\}$. Since $\textsc{ldom}\,R \subseteq \textsc{rdom}\,f$, we know that $\forall y.\ y \in \textsc{ldom}\,R \Rightarrow y \in \textsc{rdom}\,f$; on the other hand, we also have $y \in \textsc{rdom}\,f \Rightarrow \exists x.\ x \in X$. Therefore, $\forall y.\ y \in \textsc{ldom}\,R \Rightarrow \exists x.\ x \in X$ and thus $\textsc{rdom}\,(f \cdot R) = \{\, z \in Z \mid \exists y \in Y,\ \exists\, x \in X,\ x\ f\ y\ R\ z \,\} = \{\, z \in Z \mid \exists y \in Y,\ y\ R\ z \,\} = \textsc{rdom}\,R$. □

Now, we present the main proof:

$$\begin{aligned}
&\mathit{fst} \cdot \mathit{ls}_{ac} \\
=&\{\ R = R \cdot \mathit{id}_{\textsc{rdom}\,R}\ \} \\
&\mathit{fst} \cdot \mathit{ls}_{ac'} \cdot \mathit{id}_{\textsc{rdom}\,(\mathit{ls}_{ac'})} \\
\subseteq&\{\ \mathit{id}_{\textsc{rdom}\,(\mathit{ls}_{ac'})} \subseteq \mathit{ls}_{c'b'} \cdot \mathit{ls}^{\circ}_{c'b'} \text{ by sub-proof-1 below}\ \} \\
&\mathit{fst} \cdot \mathit{ls}_{ac'} \cdot (\mathit{ls}_{c'b'} \cdot \mathit{ls}^{\circ}_{c'b'}) \\
=&\{\ \text{Relation composition is associative}\ \} \\
&\mathit{fst} \cdot (\mathit{ls}_{ac'} \cdot \mathit{ls}_{c'b'}) \cdot \mathit{ls}^{\circ}_{c'b'} \\
=&\{\ \mathit{ls}_{ab'} = \mathit{ls}_{ac'} \cdot \mathit{ls}_{c'b'}\ (⑥ \text{ in Figure 5.3})\ \} \\
&\mathit{fst} \cdot \mathit{ls}_{ab'} \cdot \mathit{ls}^{\circ}_{c'b'} \\
\subseteq&\{\ \text{Retentiveness of } \mathit{lens}_{AB} \text{ and } \mathit{ls}^{\circ}_{c'b'} = \mathit{ls}_{b'c'}\ \} \\
&\mathit{fst} \cdot \mathit{ls}_{a'b'} \cdot \mathit{ls}_{b'c'} \\
=&\{\ \mathit{ls}_{a'c'} = \mathit{ls}_{a'b'} \cdot \mathit{ls}_{b'c'}\ \} \\
&\mathit{fst} \cdot \mathit{ls}_{a'c'}\ .
\end{aligned}$$

sub-proof-1: $\mathit{id}_{\textsc{rdom}\,(\mathit{ls}_{ac'})} \subseteq \mathit{ls}_{c'b'} \cdot \mathit{ls}^{\circ}_{c'b'} \Leftrightarrow \textsc{rdom}\,(\mathit{ls}_{ac'}) \subseteq \textsc{ldom}\,(\mathit{ls}_{c'b'})$ and we prove the latter using linear proofs. The right column of each line gives the reason how it is derived.

1.	$\textsc{ldom}\,(\mathit{ls}_{c'b'}) = \textsc{rdom}\,(\mathit{ls}_{b'c'})$	definition of relations
2.	$\mathit{fst} \cdot \mathit{ls}_{bc'} \subseteq \mathit{fst} \cdot \mathit{ls}_{b'c'}$	Retentiveness of lens_{BC}

3.	$\textsc{rdom}\,(fst \cdot ls_{bc'}) \subseteq \textsc{rdom}\,(fst \cdot ls_{b'c'})$	2 and definition of relation inclusion
4.	$\textsc{rdom}\,(ls_{bc'}) \subseteq \textsc{rdom}\,(ls_{b'c'})$	3 and Lemma A.1.1
5.	$ls_{bc'} = (ls^{\circ}_{ac'} \cdot ls_{ab})^{\circ} = (ls_{c'a} \cdot ls_{ab})^{\circ}$	③ in Figure 5.3
6.	$\textsc{rdom}\,(ls_{bc'}) = \textsc{rdom}\,(ls_{c'a} \cdot ls_{ab})^{\circ}$	5
7.	$\textsc{rdom}\,(ls_{c'a} \cdot ls_{ab})^{\circ} = \textsc{ldom}\,(ls_{c'a} \cdot ls_{ab})$	definition of converse relation
8.	$\textsc{ldom}\,(ls_{c'a} \cdot ls_{ab}) \subseteq \textsc{ldom}\,(ls_{c'a})$	definition of relation composition
9.	$\textsc{ldom}\,(ls_{c'a}) = \textsc{rdom}\,(ls_{ac'})$	definition of converse relation
10.	$\textsc{rdom}\,(ls_{bc'}) = \textsc{rdom}\,(ls_{ac'})$	6, 7, 8, and 9
11.	$\textsc{rdom}\,(ls_{ac'}) \subseteq \textsc{ldom}\,(ls_{c'b'})$	10, 4, and 1

□

A.2 Retentiveness of the DSL

In this section, we prove that the *get* and *put* semantics given in Section 5.2.3 does satisfy the three properties (Definition 5.1.5) of a retentive lens. Most of the proofs are proved by induction on the size of the trees.

Lemma A.2.1. The *get* function described in Section 5.2.3.1 is total.

Proof. Because we require source pattern coverage, *get* is defined for all the input data. Besides, since our DSL syntactically restricts source pattern $spat_k$ to not being a bare variable pattern, for any $v \in Vars(spat_k)$, $decompose(spat_k, s)$ is a proper subtree of s. So the recursion always decreases the size of the s parameter and thus terminates. □

Lemma A.2.2. For a pair of *get* and *put* described in Section 5.2.3.2 and any $s : S$, $check(s, get(s)) = True$.

Proof. We prove the lemma by induction on the structure of s. By the definition of *get* and *check*,

$$
\begin{aligned}
& check(s, get(s)) \\
=& \{\ get(s) \text{ produces consistency links}\ \} \\
& chkWithLink(s, get(s)) \\
=& \{\ \text{Unfolding } get(s)\ \} \\
& chkWithLink(s, reconstruct(vpat_k, fst \circ vls), l_{root} \cup links)
\end{aligned}
$$

where $vpat_k$, fst, vls, l_{root} and $links$ are those in the definition of get (5.5). In *chkWithLink*, $cond_1$ and $cond_2$ are true by the evident semantics of pattern matching functions such as *isMatch* and *reconstruct*. $cond_3$ is true following the definition of l_{root}, $links$, and *divide*. Finally, $cond_4$ is true by the inductive hypothesis. □

Lemma A.2.3. (Focusing) If $sel(s, p) = s'$ and for any $((\_, spath), (\_, \_)) \in ls$, p is a prefix of $spath$, then

$$put(s, v, ls) = put(s', v, ls') \text{ and } check(s, v, ls) = check(s', v, ls')$$

where $ls' = \{ ((a, b), (c, d)) \mid ((a, p +\!\!+ b), (c, d)) \}$.

Proof. From the definitions of *put* and *check*, we find that their first argument (of type S) is invariant during the recursive process. In fact, the first argument is only used when checking whether a link in ls is valid with respect to the source tree. Since all links in ls connect to the subtree s', the parts in s above s' can be trimmed and the identity holds. □

Theorem A.2.4. (Hippocraticness of the DSL) For any s of type S,

$$put(s, get(s))^{1} = s \ .$$

Proof of Hippocraticness. Also by induction on the structure of s,

$$\begin{aligned} & put(s, get(s)) \\ =& \{ \text{ Unfolding } get(s) \ \} \\ & put(s, reconstruct(vpat_k, fst \circ vls), l_{root} \cup links) \ , \end{aligned}$$

where $spat_k \sim vpat_k \in R$ is the unique rule such that $spat_k$ matches s. l_{root}, $links$, and vls are defined exactly the same as in get (5.5).

Now we expand put. Because l_{root} links to the root of the view, put falls to its second case.

$$put(s, get(s)) = inj(reconstruct(spat_k', ss)) \tag{A.1}$$

[1] For simplicity, we regard $(a, (b, c))$ the same as (a, b, c).

where

$$
\begin{aligned}
&spat_k{}' \\
&= \{\ spat \text{ in (5.9) is } eraseVars(fillWildcards(spat_k, s))\ \} \\
&fillWildcards(spat_k, eraseVars(fillWildcards(spat_k, s))) \\
&= \{\ \text{See Figure A.1}\ \} \\
&fillWildcards(spat_k, s)\ .
\end{aligned}
$$

and

$$
ss = \lambda(t \in Vars(spat_k)) \longrightarrow put(s, vs(t), divide(Path(vpat_k, t), links))
$$

where $vs = decompose(vpat_k, reconstruct(vpat_k, fst \circ vls)) = fst \circ vls$. (See the beginning of the proof.) Since $vls = get \circ decompose(spat_k, s)$, we have

$$
\begin{aligned}
ss = \lambda(t \in Vars(spat_k)) \longrightarrow \\
put(s, fst(get(decompose(spat_k, s)(t))), divide(Path(vpat_k, t), links))
\end{aligned}
$$

By Lemma A.2.3, we have

$$
\begin{aligned}
ss &= \lambda(t \in Vars(spat_k)) \longrightarrow \\
&\qquad put(decompose(spat_k, s)(t), fst(get(decompose(spat_k, s)(t))), \\
&\qquad\quad snd(get(decompose(spat_k, s)(t)))) \\
&= \{\ \text{Inductive hypothesis for } decompose(spat_k, s)(t)\ \} \\
&\quad \lambda(t \in Vars(spat_k)) \longrightarrow decompose(spat_k, s)(t) \\
&= decompose(spat_k, s)\ .
\end{aligned}
$$

Now, we substitute $fillWildcards(spat_k, s)$ for $spat_k{}'$ and $decompose(spat_k, s)$ for ss in equation (A.1), and obtain

$$
\begin{aligned}
&put(s, get(s)) \\
=&\{\ Equation(A.1)\ \} \\
&inj_{S \to S@V}(reconstruct(spat_k{}', ss)) \\
=&inj_{S \to S@V}(reconstruct(fillWildcards(spat_k, s), decompose(spat_k, s)))
\end{aligned}
$$

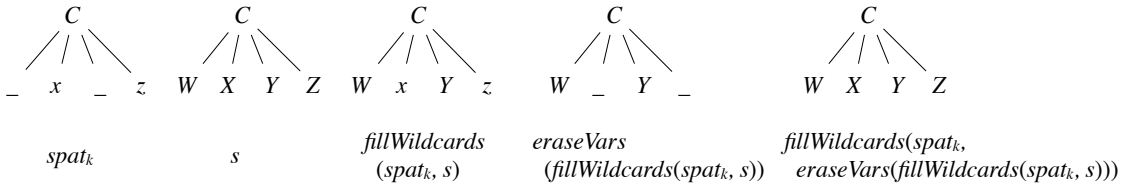


Figure A.1: A property regarding *fillWildcards*.

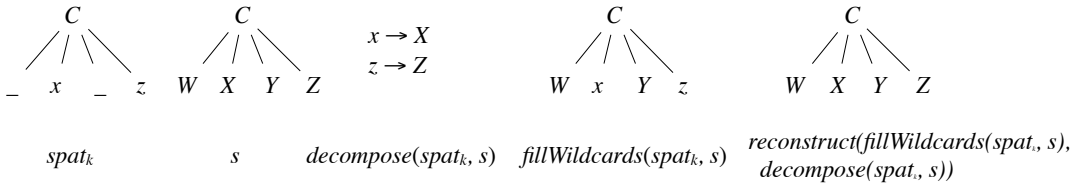


Figure A.2: A property regarding Reconstruct-Decompose.

$$
\begin{aligned}
&=\{ \text{ See Figure A.2 } \}\\
&\quad inj_{S \to S@V}(s)\\
&= s\ .
\end{aligned}
$$

This completes the proof of Hippocraticness. □

Theorem A.2.5. (Correctness of the DSL) For any (s, v, ls) that makes $check(s, v, ls) = True$, $get(put(s, v, ls)) = (v, ls')$, for some ls'.

Proof of Correctness. We prove Correctness by induction on the size of (v, ls). The proofs of the two cases of *put* are quite similar, and therefore we only present the first one, in which $put(s, v, ls)$ falls into the first case of *put*: i.e.

$$put(s, v, ls) = reconstruct(fillWildcardsWithDefaults(spat_k), ss)\ .$$

Then

$$get(put(s, v, ls)) = get(reconstruct(fillWildcardsWithDefaults(spat_k), ss))$$

where $spat_k \sim vpat_k \in R$, $isMatch(vpat_k, v) = True$, and

$$ss = \lambda(t \in Vars(spat_k)) \rightarrow put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls)) \ .$$

Now expanding the definition of *get*, because of the disjointness of source patterns, the same $spat_k \sim vpat_k \in R$ will be select again. Thus

$$get(put(s, v, ls)) = (reconstruct(vpat_k, fst \circ vls), \cdots)$$

where

$$\begin{aligned}
vls &= get \circ decompose(spat_k, put(s, v, ls)) \\
&= get \circ decompose(spat_k, reconstruct(fillWildcardsWithDefaults(spat_k), ss)) \\
&= \{ \text{ See Figure A.3 } \} \\
&\ get \circ ss \\
&= \lambda(t \in Vars(spat_k)) \rightarrow get(put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls)))
\end{aligned}$$

To proceed, we want to use the inductive hypothesis to simplify $get(put(\cdots))$. When $vpat_k$ is not a bare variable pattern, $decompose(vpat_k, v)(t)$ is a proper subtree of v and the size of the third argument (i.e. links ls) is non-increasing; thus the inductive hypothesis is applicable. On the other hand, if $vpat_k$ is a bare variable pattern, the sizes of all the arguments stays the same; but $cond_3$ in *chkNoLink* guarantees that in the next round of the recursion, a pattern $vpat_k$ that is not a bare variable pattern will be selected. Therefore we can still apply the inductive hypothesis. Applying the inductive hypothesis, we get

$$vls = \lambda(t \in Vars(spat_k)) \rightarrow (decompose(vpat_k, v)(t), \cdots)$$

Thus $get(put(s, v, ls)) = (reconstruct(vpat_k, decompose(vpat_k, v)), \cdots) = (v, \cdots)$, which completes the proof of Correctness. □

Theorem A.2.6. (Retentiveness of the DSL) For any (s, v, ls) that $check(s, v, ls) = True$, $get(put(s, v, ls)) = (v', ls')$, for some v' and ls' such that

$$\{ (spat, (vpat, vpath)) \mid ((spat, spath), (vpat, vpath)) \in ls \}$$

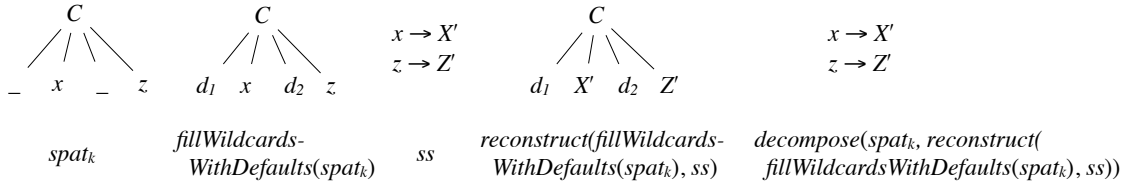


Figure A.3: A property regarding Decompose-Reconstruct.

$$\subseteq \{\, (spat, (vpat, vpath)) \mid ((spat, spath),\ (vpat, vpath)) \in ls' \,\}$$

Proof of Retentiveness. Again, we prove Retentiveness by induction on the size of (v, ls). The proofs of the two cases of *put* are similar, and thus we only show the second one here.

If there is some $l = ((spat, spath),\ (vpat, [\,])) \in ls$, let $spat_k \sim vpat_k$ be the unique rule in $S \sim V$ that $isMatch(spat_k, spat) = True$. We have

$$\begin{aligned}
&get(put(s, v, ls)) \\
=&\{ \text{ Definition of } put \,\} \\
&get(inj_{TypeOf(spat_k) \to S}(s')) \\
=&\{ \, get(inj(s)) = get(s) \text{ as shown in (Section 5.2.2.1)} \,\} \\
&get(s')
\end{aligned}$$

where $s' = reconstruct(fillWildcards(spat_k, spat), ss)$ and

$$\begin{aligned}
ss = \lambda(t \in Vars(spat_k)) \to \\
put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls \setminus \{\, l \,\}))
\end{aligned}$$

Now we expand the definition of *get* (and focus on the links)

$$get(put(s, v, ls)) = (\cdots, \{\, l_{root} \,\} \cup links)$$

where $l_{root} = ((eraseVars(fillWildcards(spat_k, s')), [])\,,\ (eraseVars(vpat_k), []))$,

$$\begin{aligned}
links = \{\, ((a, Path(spat_k, t) \mathbin{+\!\!+} b),\ (c, Path(vpat_k, t) \mathbin{+\!\!+} d)) \\
\mid t \in Vars(vpat_k), ((a, b),\ (c, d)) \in snd(vls(t)) \,\} \text{ ,and}
\end{aligned} \tag{A.2}$$

$$
\begin{aligned}
& vls(t) \\
=& \{ \text{ Unfolding } vls \ \} \\
& (get \circ decompose(spat_k, s'))(t) \\
=& \{ \text{ Unfolding } s' \ \} \\
& (get \circ decompose(spat_k, reconstruct(fillWildcards(spat_k, spat), ss)))(t) \\
=& \{ \text{ Similar to the case shown in Figure A.3 } \} \\
& (get \circ ss)(t) \\
=& \{ \text{ Definition of } ss \ \} \\
& get(put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls \setminus \{ l \}))) \ .
\end{aligned}
$$

For l_{root}, we have

$$
\begin{aligned}
& eraseVars(fillWildcards(spat_k, s')) \\
=& \{ \text{ Unfolding } s' \ \} \\
& eraseVars(fillWildcards(spat_k, reconstruct(fillWildcards(spat_k, spat), ss))) \\
=& \{ \text{ See Figure A.4 } \} \\
& eraseVars(fillWildcards(spat_k, spat)) \\
=& \{ \text{ By } cond_2 \text{ in } chkWithLink \ \} \\
& spat
\end{aligned}
$$

Use the first clause of $cond_2$, we have $vpat = eraseVars(vpat_k)$. Thus

$$
l_{root} = ((eraseVars(fillWildcards(spat_k, s')), []) \ , \ (eraseVars(vpat_k), [])) = ((spat, [\,]), \ (vpat, [\,])) \ ,
$$

and therefore the input link $l = ((spat, spath), (vpat, [\,]))$ is 'preserved' by l_{root}, i.e. $fst \cdot \{l\} = fst \cdot \{l_{root}\}$.

For the links in $ls \setminus \{ l \}$, we show that they are preserved in *links* (A.2) above. By $cond_3$ in *chkWithLink*, for every link $m \in ls \setminus \{ l \}$, there is some t_m in $Vars(spat_k)$ such that

$$
m \in addVPrefix(Path(vpat_k, t_m), divide(Path(vpat_k, t_m), ls \setminus \{ l \})).
$$

If $m = ((a, b), (c, Path(vpat_k, t_m) \mathbin{+\!\!+} d))$, then

$$
m' = ((a, b), \ (c, d)) \in divide(Path(vpat_k, t_m), ls \setminus \{ l \}).
$$

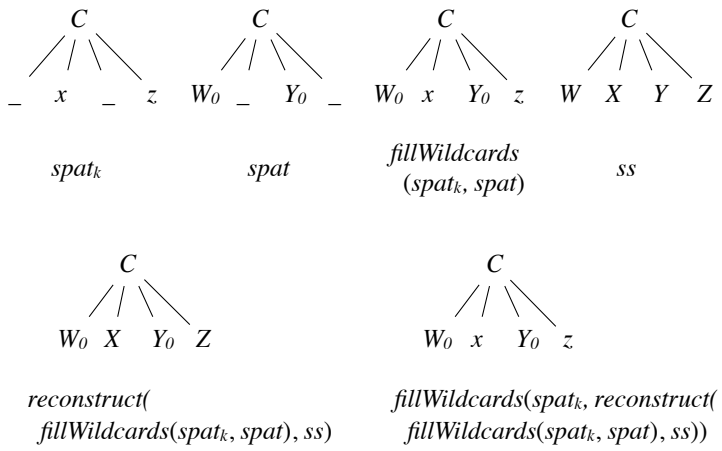


Figure A.4: Another property regarding *fillWildcards*.

By the inductive hypothesis for $snd(vls(t_m))$, m' is 'preserved', that is

$$\exists b'.\ ((a, b'),\ (c, d)) \in snd(vls(t_m))$$

Now by the definition of *links* (A.2), $((a, Path(spat_k, t_m) \nplus b'), (c, Path(vpat_k, t_m) \nplus d)) \in links$, therefore m is also preserved. □

Corollary A.2.7. Let $put' = put$ with its domain intersected with $S \times V \times LinkSet$, *get* and put' form a retentive lens as in Definition 5.1.5 since they satisfy Hippocraticness (5.1), Correctness (5.2) and Retentiveness (5.3).

B

Refactoring Operations as Edit Operation Sequences

We summarise how the 23 refactoring operations for Java 8 in Eclipse Oxygen could be described by *replace*, *copy*, *move*, *swap*, *insert*, and *delete*, where the *insert* and *delete* operations on lists can be implemented in terms of the first four. For instance, to *insert* an element e at position i in a list of length n (where $1 \leqslant i \leqslant n$), we can follow these steps: (i) Change the list to length $n + 1$. (ii) Starting from the tail of the list, *move* each element at position j such that $j > i$ to position $j + 1$. (iii) *replace* the element at position i with e. Deleting the element at position i is almost as simple as moving each element after i one position ahead and decrease the length of the list by one.

Table B.1: Refactoring Operations as Edit Operation Sequences.

Refactor Operation	Description	Edit Operations
Rename	Renames the selected element and (if enabled) corrects all references to the elements	*replace* the selected element and all references with the new name.

Use Supertype Where Possible	Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible.	*replace* all occurrences.
Generalize Declared Type	Allows the user to choose a supertype of the reference's current type. If the reference can be safely changed to the new type, it is.	*replace* all occurrences.
Infer Generic Type Arguments	Replaces raw type occurrences of generic types by parameterized types after identifying all places where this replacement is possible.	*replace* all occurrences.
Encapsulate Field	Replaces all references to a field with getter and setter methods.	*insert* getters and setters; *replace* all occurrences (with getters or setters respectively).
Change Method Signature	Changes parameter names, parameter types, parameter order and updates all references to the corresponding method.	*replace* all occurrences. Use *swap* if we need to change the parameter order.
Extract Method	Creates a new method containing the statements or expression currently selected and replaces the selection with a reference to the new method.	*insert* a new method; *move* selected code; *replace* the selection.
Extract Local Variable	Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.	*insert* a new variable; *copy* the selected expression to the variable assignment; *replace* the selected expression.
Extract Constant	Creates a static final field from the selected expression and substitutes a field reference, and optionally rewrites other places where the same expression occurs.	*insert* a field; *copy* the selected expression; *replace* the selected expression.
Introduce Parameter	Replaces an expression with a reference to a new method parameter, and updates all callers of the method to pass the expression as the value of that parameter.	*insert* a method parameter; *insert* the selected expression to all the callers (use *copy* if we want to preserve the information attached to the expression); *replace* the expression with the new method parameter.
Introduce Factory	Creates a new factory method, which will call a selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method.	*insert* a factory method; *replace* all the references to the constructor.

Introduce Indirection	Creates a static indirection method delegating to the selected method.	*insert* a method.
Convert to Nested	Converts an anonymous inner class to a member class.	*insert* a member class; *move* the code within the anonymous class to the member class; *delete* the anonymous class.
Move Type to New File	Creates a new Java compilation unit for the selected member type or the selected secondary type, updating all references as needed.	*Move* the selected code to the new file; *replace* all references.
Convert Local Variable to Field	Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors.	*insert* a field; *copy* the initialization; *delete* the variable declaration.
Extract Superclass	Extracts a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring.	*insert* a superclass; *move* fields to the superclass; *replace* declarations of sibling types (classes) so that they extend the superclass; *insert* lacking fields into sibling classes.
Extract Interface	Creates a new interface with a set of methods and makes the selected class implement the interface.	generally the same as above.
Move	Moves the selected elements and (if enabled) corrects all references to the elements (also in other files).	*move* the selected elements; *replace* all references.
Push Down	Moves a set of methods and fields from a class to its subclasses.	*move* the methods and fields.
Pull Up	Moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass.	*move* the field or *insert* an abstract method declaration.
Introduce Parameter Object	Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduce parameter.	*insert* a class definition; *move* the parameters to the class; in callers' definitions, *delete* the set of parameters and *insert* the class type as a new parameter; for callers' arguments, *delete* the arguments corresponding to the set of parameters and *insert* a class instance.

Extract Class	Replaces a set of fields with new container object. All references to the fields are updated to access the new container object.	*insert* a new class; *move* the fields; *replace* references to the fields with references to the container object and field names.
Inline	Inline local variables, methods or constants.	For a variable or a constant, *replace* the occurrences with the value; *delete* the definition. For a method, *replace* all occurrences of parameters within the method body with real arguments; *replace* the method call with the (new) method body; *delete* the method definition.