New Generation Computing
https://doi.org/10.1007/s00354-019-00082-y

Unifying Parsing and Reflective Printing for Fully Disambiguated Grammars

Zirun Zhu[1] · Hsiang-Shang Ko[2] · Yongzhe Zhang[1] · Pedro Martins[3] · João Saraiva[4] · Zhenjiang Hu[5]

Received: 31 January 2019 / Accepted: 16 December 2019

Abstract
Language designers usually need to implement parsers and printers. Despite being two closely related programs, in practice they are often designed separately, and then need to be revised and kept consistent as the language evolves. It will be more convenient if the parser and printer can be unified and developed in a single program, with their consistency guaranteed automatically. Furthermore, in certain scenarios (like showing compiler optimisation results to the programmer), it is desirable to have a more powerful reflective printer that, when an abstract syntax tree corresponding to a piece of program text is modified, can propagate the modification to the program text while preserving layouts, comments, and syntactic sugar. To address these needs, we propose a domain-specific language BiYacc, whose programs denote both a parser and a reflective printer for a fully disambiguated context-free grammar. BiYacc is based on the theory of bidirectional transformations, which helps to guarantee by construction that the generated pairs of parsers and reflective printers are consistent. Handling grammatical ambiguity is particularly challenging: we propose an approach based on generalised parsing and disambiguation filters, which produce all the parse results and (try to) select the only correct one in the parsing direction; the filters are carefully bidirectionalised so that they also work in the printing direction and do not break the consistency between the parsers and reflective printers. We show that BiYacc is capable of facilitating many tasks such as Pombrio and Krishnamurthi's 'resugaring', simple refactoring, and language evolution.

Keywords Asymmetric lenses · Disambiguation filters · Bidirectional transformations · Domain-specific languages · Parsing · Reflective printing

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00354-019-00082-y) contains supplementary material, which is available to authorized users.

Extended author information available on the last page of the article

Introduction

Whenever we come up with a new programming language, as the front-end part of the system we need to design and implement a parser and a printer to convert between program text and an internal representation. 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 more structured and simplified representation and is easier for the back-end to manipulate. On the other hand, a printer converts an AST back to a piece of program text, which can be understood by the user of the system; this is useful for debugging the system, or reporting internal information to the user.

Parsers and printers do conversions in opposite directions and are closely related—for example, the program text printed from an AST should be parsed to the same tree. It is certainly far from being economical to write parsers and printers separately: the parser and printer need to be revised from time to time as the language evolves, and each time we must revise the parser and printer and also keep them consistent with each other, which is a time-consuming and error-prone task. In response to this problem, many domain-specific languages [6, 7, 13, 37, 44, 53] have been proposed, in which the user can describe both a parser and a printer in a single program.

Despite their advantages, these domain-specific languages cannot deal with synchronisation between program text and ASTs. Let us look at a concrete example in Fig. 1: 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 idea is to propagate the changes 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 likely mislead the programmer into thinking that the negation is replaced by a subtraction by the compiler; also, since the comment is not preserved, it will be harder for the

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 *reflective* printer:

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

Fig. 1 Comparison between conventional printing and reflective printing

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | ☐ LE | ☐ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

programmer to compare the updated and original versions of the text. The problem illustrated here has also been investigated in many other practical scenarios where the parser and printer are used as a bridge between the system and the user, for example,

- in bug reporting [51], where a piece of program text is parsed to its AST to be checked but error messages should be displayed for the program text;
- in code refactoring [18], 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; and
- in language-based editors, as introduced by Reps [45, 46], where the user needs to interact with different printed representations of the same underlying AST.

To address the problem, we propose a domain-specific language BiYacc, which enables the user to describe both a parser and a reflective printer for a fully disambiguated context-free grammar (CFG) in a single program. Different from a conventional printer, a reflective printer takes a piece of program text and an AST, which is usually slightly modified from the AST corresponding to the original program text, and propagates the modification back to the program text. Meanwhile the comments (and layouts) in the unmodified parts of the program text are all preserved. This can be seen clearly from the result of using our reflective printer on the above arithmetic expression example in Fig. 1. It is worth noting that reflective printing is a generalisation of the conventional notion of printing, because a reflective 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.

From a BiYacc program, we can generate a parser and a reflective printer; in addition, we want to guarantee that the two generated components are consistent with each other. Specifically, given a pair of parser *parse* and reflective printer *print*, we want to ensure two (inverse-like) consistency properties: first, a piece of program text s printed from an abstract syntax tree t should be parsed to the same tree t, i.e.[1]

$$\mathit{parse}\ (\mathit{print}\ s\ t) = t\ . \tag{1}$$

Second, updating a piece of program text s with an AST parsed from s should leave s unmodified (including formatting details like parentheses and whitespaces), i.e.

$$\mathit{print}\ s\ (\mathit{parse}\ s) = s\ . \tag{2}$$

[1] We assume basic knowledge about functional programming languages and their notations, in particular Haskell [5, 34]. 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, $\mathit{parse}\ (\mathit{print}\ s\ t) = t$ is in fact $\forall s\ t.\ \mathit{parse}\ (\mathit{print}\ s\ t) = t$.

Springer Ohmsha

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

These two properties are inspired by the theory of bidirectional transformations [19], in particular lenses [17], and are guaranteed by construction for all BiYacc programs.

An online tool that implements the approach described in the paper can be accessed at http://www.prg.nii.ac.jp/project/biyacc.html. The webpage also contains the test cases used in the paper. The structure of the paper is as follows: we start with an overview of BiYacc in Sect. 2, explaining how to describe in a single program both a parser and a reflective printer for synchronising program text and its abstract syntax representation. After reviewing some background on bidirectional transformations in Sect. 3, in particular the bidirectional programming language BiGUL [22, 27, 28], we first give the semantics of a basic version of BiYacc that handles unambiguous grammars by compiling it to BiGUL in Sect. 4, guaranteeing the properties (1) and (2) by construction. Then, inspired by the research on generalised parsing [50] and disambiguation filters [26], in Sect. 5 we revise the basic BiYacc architecture to allow the use of ambiguous grammars and disambiguation directives while still retaining the above-mentioned properties. We present a case study in Sect. 6, showing that BiYacc is capable of describing Tiger [4], which shares many similarities with fully fledged languages. We demonstrate that BiYacc can handle syntactic sugar, partially subsume Pombrio and Krishnamurthi's 're-sugaring' [42, 43], and facilitate language evolution. In Sect. 7, we present detailed related work including comparison with other systems. Contributions are summarised in Sect. 8.

This is the extended version of our previous work Parsing and Reflective Printing, Bidirectionally presented at SLE'16 [55], and the differences are mainly as follows: (1) we propose the notion of bidirectionalised filters and integrate them into BiYacc for handling grammatical ambiguity (Sect. 5); the related work section is also updated accordingly. (2) We restructure the narration for introducing the basic BiYacc system and in particular elaborate on the isomorphism between program text and CSTs. (3) We present the definitions and theorems in a more formal way, and complete their proofs. (4) We make several other revisions such as renewing the figures for introducing the BiYacc system and the syntax of BiYacc programs.

Throughout this paper, we typeset general definitions and properties in *math style* and specific examples in `code style`.

A First Look at BiYacc

We first give an overview of BiYacc by going through the BiYacc program shown in Fig. 2, which deals with the arithmetic expression example given in Sect. 1. This program consists of definitions of the abstract syntax, concrete syntax, directives, and actions for reflectively printing ASTs to CSTs; we will introduce them in order.

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

```
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 ;;
```

Fig. 2 A BiYacc program for the expression example

syntax is defined in lines 2–7 by a single data type 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 "//",[2] 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 Fig. 6 needs to be augmented with a few such directives).

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | ☐ LE | ☐ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 Fig. 2. 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. We will expound on this identification of program text with CSTs in Sect. 4.2 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 updated instructions, which are also denoted by '+>'.

Let us look at a specific action—the first one for the expression example, at line 28 of Fig. 2:

```
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 Sect. 4.

Deep patterns Using deep patterns, we can write actions that establish nontrivial relationships between CSTs and ASTs. For example, the action at line 38 of Fig. 2 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 Sect. 6.

Layout and comment preservation The reflective printer generated by BiYacc is capable of preserving layouts and comments, but, perhaps mysteriously, in Fig. 2 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.

Parsing semantics So far, we have been describing the reflective 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 [23] actions.) In fact, this paper will not define the parsing semantics formally, because the parsing semantics is completely determined by the reflective printing semantics: 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. We will explain how this is achieved in the next section.

Foundation of BiYacc: Putback-Based Bidirectional Programming

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. satisfy the properties (1) and (2) stated in Sect. 1. It is possible to implement the *print* and *parse* semantics separately in an ad hoc way, but verifying the two consistency properties takes extra effort. The implementation we present, however, is systematic and guarantees consistency by construction, thanks to the well-developed theory of bidirectional transformations (BXs for short), in particular lenses [17]. We will give a brief introduction to BXs below; for a comprehensive treatment, the readers are referred to the lecture notes for the 2016 Oxford Summer School on Bidirectional Transformations [19].

Parsing and Printing as Lenses

The *parse* and *print* semantics of BiYacc programs are potentially partial—for example, if the actions in a BiYacc program do not cover all possible forms of program text and abstract syntax trees, *parse* and *print* will fail for those uncovered inputs. Thus, we should take partiality into account when choosing a BX framework in which to model *parse* and *print*. The framework we use in this paper is an explicitly partial version [32, 40] of asymmetric lenses [17].

Definition 1 (*Lenses*) A *lens* between a source type S and a view type V is a pair of functions

$$\begin{aligned} get &:: S \rightarrow \mathsf{Maybe}\ V \\ put &:: S \rightarrow V \rightarrow \mathsf{Maybe}\ S \end{aligned}$$

satisfying the well-behavedness laws:

$$\begin{array}{llll} put\ s\ v = \mathsf{Just}\ s' & \Rightarrow & get\ s' \ = \mathsf{Just}\ v & (\textsc{PutGet}) \\ get\ s \ = \mathsf{Just}\ v & \Rightarrow & put\ s\ v = \mathsf{Just}\ s & (\textsc{GetPut}) \end{array}$$

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. Partiality is explicitly represented by making the functions return Maybe values: a *get* or *put* function returns Just r where r is the result, or Nothing if the input is not in the domain. The PutGet 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 GetPut 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.

The *parse* and *print* semantics of a BiYacc program will be the pair of functions *get* and *put* in a lens, required by definition to satisfy the two well-behavedness laws, which are exactly the consistency properties (1) and (2) reformulated in a partial setting:

Definition 2 (*The Partial Version of Consistency Properties*)

$$\begin{array}{llll} print\ s\ t \ = \mathsf{Just}\ s' & \Rightarrow & parse\ s' \ = \mathsf{Just}\ t \\ parse\ s \ = \mathsf{Just}\ t & \Rightarrow & print\ s\ t \ = \mathsf{Just}\ s \end{array}$$

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

Putback-Based Bidirectional Programming in BiGUL

Having rephrased parsing and printing in terms of lenses, we can now construct consistent pairs of parsers and printers using bidirectional programming techniques, in which the programmer writes a single program to denote the two directions of a lens. Specifically, BiYacc programs are compiled to the putback-based bidirectional programming language BiGUL [28]. It has been formally verified in Agda [39] that BiGUL programs always denote well-behaved lenses, and BiGUL has been ported to Haskell as an embedded DSL library [22]. BiGUL is putback-based, meaning that a BiGUL program describes a *put* function, but—since BiGUL is bidirectional—can also be executed as the corresponding *get* function. The advantage of putback-based bidirectional programming lies in the fact that, given a *put* function, there is at most one *get* function that forms a (well-behaved) lens with this *put* function [16]. That is, once we describe a *put* function as a BiGUL program, the *get* semantics of the program is completely determined by its *put* semantics. We can therefore focus solely on the printing (*put*) behaviour, leaving the parsing (*get*) behaviour only implicitly (but unambiguously) specified. How the programmer can effectively work with this paradigm has been more formally explained in terms of a Hoare-style logic for BiGUL [27].

Compilation of BiYacc to BiGUL (Sect. 4) uses only three BiGUL operations, which we briefly introduce here; more details can be found in the lecture notes on BiGUL programming [22]. A BiGUL program has type $\mathsf{BiGUL}\, s\, v$, where s and v are, respectively, the source and view types.

Replace The simplest BiGUL operation we use is

$$\mathsf{Replace} :: \mathsf{BiGUL}\, s\, s$$

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 $\mathsf{Replace}$ is the function $\lambda\ s\ v \rightarrow \mathsf{Just}\ v$.

Update The next operation update is more complex, and is implemented with the help of Template Haskell [49]. The general form of the operation is

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

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

$$\$(\mathsf{update}\ [\mathsf{p}|\ (\mathsf{x}, \_)\ |]\ [\mathsf{p}|\ \mathsf{x}\ |]\ [\mathsf{d}|\ \mathsf{x} = \mathsf{Replace}\ |])$$

matches the source with a tuple pattern $(\mathsf{x}, \_)$ 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 $\mathsf{x} = \mathsf{Replace}$. (The part marked by underscore (_) simply

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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:

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

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′*:

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

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

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

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.

The Basic BiYacc

In this section, we expound on a basic version of BiYacc that handles only unambiguous grammars. (Section 5 will present extensions for dealing with ambiguous grammars with disambiguation.) The architecture is illustrated in Fig. 3, where a BiYacc program

$$\text{‘\#Abstract’}\ \mathit{decls}\ \text{‘\#Concrete’}\ \mathit{pgs}\ \text{‘\#Directives’}\ \mathit{drctvs}\ \text{‘\#Actions’}\ \mathit{ags}\ , \qquad (3)$$

consisting of abstract syntax, concrete syntax, directives, and printing actions, as formally defined in Fig. 4, is compiled into a few Haskell source files and then into

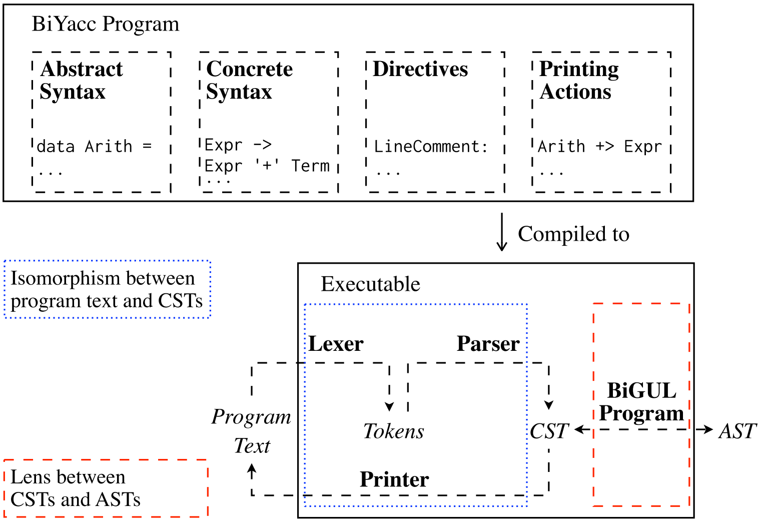

Fig. 3 Architecture of BiYacc

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*$^+$ '`)`'

Fig. 4 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 Sect. 5)

Springer Ohmsha

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 an (partial) isomorphism (which will be defined in Sect. 4.1). This part will be explained in Sect. 4.2.
- 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 1) for handling (the semantic part of) parsing and reflective printing between CSTs and ASTs. This part will be explained in Sect. 4.3.

The whole executable is a well-behaved lens since it is the composition of an isomorphism and a lens. We will start from a recap of this fact.

Composition of Isomorphisms and Lenses

First, we give the definition of (partial) isomorphisms.

Definition 3 (Isomorphism) A (partial) isomorphism between two types A and B is a pair of functions:

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

such that the inverse properties hold:

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

Definition 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

$$\begin{aligned} &\mathit{get}' && :: A \to \mathsf{Maybe}\ C \\ &\mathit{get}'\ a && = \mathit{to}\ a \gg\!= \mathit{get} \end{aligned}$$

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 [17]—an isomorphism can be lifted to a lens (with *get* s = *to* s and *put* s v = *from* v), which can then be composed with another lens to give rise to a new lens. We thus have the following lemma.

Lemma 1 *Any lens resulted from the composition in Definition 4 is well-behaved.*

Therefore the whole BiYacc executable is a well-behaved lens, given that the concrete parser and printer form an isomorphism (Theorem 1) and the BiGUL program is a well-behaved lens (Theorem 2), which we will see next.

The Concrete Parsing and Printing Isomorphism

In this subsection, we describe the generation of CST data types and concrete printers (Sect. 4.2.1), the generation of concrete parsers (Sect. 4.2.2), and finally the inverse properties satisfied by the concrete parsers and printers (Sect. 4.2.3).

Generating CST Data Types and Concrete Printers

The production rules in a context-free grammar dictate how to produce strings from nonterminals, and a CST can be regarded as encoding one particular way of producing a string using the production rules. In BiYacc, we represent CSTs starting from a nonterminal *nt* as an automatically generated Haskell data type 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 Fig. 2, the group of production rules from the nonterminal Factor (lines 18–21) is translated to the following Haskell data type 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                 = ""
```


Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

where Factor1 ... Factor4 are constructors corresponding to the four production rules, and 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 whitespaces appearing after a negation sign in the program text. The Factor3 case makes a call to cprtExpr :: Expr –> String, which is the printing function generated for the nonterminal Expr.

Following this idea, we define the translation from production rule groups (*pgs* in formula (3)) to datatype definitions by source-to-source compilation rules:

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

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. The auxiliary function $\textsc{con}(nt, body)$ 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,[3] 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 $\textsc{nullCon}(nt)$.

Generating Concrete Lexers and Parsers

The implementation of the concrete parser, which turns program text into CSTs, is further divided 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 nontermi-

[3] 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.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

nals 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 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 regarded 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 regarded 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 [33], which takes a BNF specification of a grammar 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.

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 Sect. 4.2.1 how to generate its datatype definition and a concrete printing function

$$cprint :: \mathsf{CST} \rightarrow \mathsf{String}.$$

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

$$cparse :: \mathsf{String} \rightarrow \mathsf{Maybe}\,\mathsf{CST},$$

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

$$\mathsf{Just} \circ cprint :: \mathsf{CST} \rightarrow \mathsf{Maybe}\,\mathsf{String}.$$

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 [33], we need to assume that they satisfy some essential properties, for we cannot control the generation process and verify those properties.

Definition 5 (Parser correctness) A parser *cparse* is correct with respect to a printer *cprint* exactly when

$$cparse\ text = \mathsf{Just}\ cst \quad \Rightarrow \quad cprint\ cst = text \tag{4}$$

$$cprint\ cst = text \quad \Rightarrow \quad \exists\, cst'.\ cparse\ text = \mathsf{Just}\ cst'\ . \tag{5}$$

To see what (4) means, recall that our CSTs, as described in Sect. 4.2.1, 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 (4). In other words, (4) 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 (5).

We also need to make an assumption about concrete printers: recall that in 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 1 (*Inverse Properties*) *If a parser cparse is correct with respect to an injective printer cprint, then cparse and* Just $\circ$ *cprint form an isomorphism, that is,*

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 (4). For the right-to-left direction, again the antecedent is equivalent to $cprint\ cst = text$, and we can invoke (5) to obtain $cparse\ text = \mathsf{Just}\ cst'$ 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

$$
\begin{aligned}
& \quad cparse\ text = \mathsf{Just}\ cst' \\
\Rightarrow & \quad \{\text{antecedent}\} \\
& \quad cparse\ (cprint\ cst) = \mathsf{Just}\ cst' \\
\Rightarrow & \quad \{(4)\} \\
& \quad cprint\ cst' = cprint\ cst \\
\Rightarrow & \quad \{cprint \text{ is injective}\} \\
& \quad cst' = cst\ .
\end{aligned}
$$

□

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 Fig. 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 second half of the Case statement, so that the 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 Sect. 2 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 update operation. For example, the first action in the Arith–Factor group

Springer Ohmsha
Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

$$
\begin{aligned}
&[\![vt\ \text{‘}\texttt{+>}\text{’}\ st\ acts]\!]_{ActionGroup} = \\
&\quad \textsc{prog}(vt, st)\ \text{‘}\texttt{::}\text{’}\ \text{‘}\texttt{BiGUL}\text{’}\ st\ vt \\
&\quad \textsc{prog}(vt, st)\ \text{‘}\texttt{=}\text{’} \\
&\quad\quad \text{‘}\texttt{Case}\text{’}\ \text{‘}\texttt{[}\text{’}\ \langle [\![a]\!]^{\mathsf{N}, vt, st}_{Action}\ \text{‘}\texttt{,}\text{’} \mid a \in acts \rangle\ \langle [\![a]\!]^{\mathsf{A}, st}_{Action} \mid a \in acts \rangle \{\text{‘}\texttt{,}\text{’}\}\ \text{‘}\texttt{]}\text{’}
\end{aligned}
$$

$$
\begin{aligned}
&[\![vpat\ \text{‘}\texttt{+>}\text{’}\ updates]\!]^{\mathsf{N}, vt, st}_{Action} = \\
&\ \text{‘}\texttt{\$(normalSV}\text{’} \\
&\quad \text{‘}\texttt{[p|}\text{’}\ \textsc{srcCond}(\textsc{ersVars}(\text{‘}\texttt{[}\text{’}\ st\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{]}\text{’})_{Update})\ \text{‘}\texttt{|]}\text{’}\ \text{‘}\texttt{[p|}\text{’}\ vpat\ \text{‘}\texttt{|]}\text{’} \\
&\quad \text{‘}\texttt{[p|}\text{’}\ \textsc{srcCond}(\textsc{ersVars}(\text{‘}\texttt{(}\text{’}\ st\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’})_{Update})\ \text{‘}\texttt{|])}\text{’} \\
&\quad\quad \text{‘}\texttt{\$(update}\text{’}\ \text{‘}\texttt{[p|}\text{’}\ \textsc{removeAs}(vpat)\ \text{‘}\texttt{|]}\text{’} \\
&\quad\quad\quad\quad\quad\quad \text{‘}\texttt{[p|}\text{’}\ \textsc{srcPat}(\text{‘}\texttt{(}\text{’}\ st\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’})_{Update}\ \text{‘}\texttt{|]}\text{’} \\
&\quad\quad\quad\quad\quad\quad \text{‘}\texttt{[d|}\text{’}\ \langle [\![u]\!]^{vt, vpat}_{Update} \mid u \in updates \rangle\ \text{‘}\texttt{|])}\text{’}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ uc_{Primitive}\ \text{‘}\texttt{]}\text{’}]\!]^{vt, vpat}_{Update} = var\ \text{‘}\texttt{= Replace;}\text{’} \\
&[\![\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ uc_{Nonterminal}\ \text{‘}\texttt{]}\text{’}]\!]^{vt, vpat}_{Update} = var\ \text{‘}\texttt{=}\text{’}\ \textsc{prog}(\textsc{varType}(vt, vpat, var), uc)\ \text{‘}\texttt{;}\text{’} \\
&[\![\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ \text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ \ldots \text{‘}\texttt{)}\text{’}\ \text{‘}\texttt{]}\text{’}]\!]^{vt, vpat}_{Update} = [\![\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ nt\ \text{‘}\texttt{]}\text{’}]\!]^{vt, vpat}_{Update} \\
&[\![\text{‘}\texttt{(}\text{’}\ \ldots\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’}]\!]^{vt, vpat}_{Update} = \langle [\![u]\!]^{vt, vpat}_{Update}\ \text{‘}\texttt{;}\text{’} \mid u \in updates \rangle \\
&[\![symbol]\!]^{vt, vpat}_{Update} = \text{‘’}
\end{aligned}
$$

$$
\begin{aligned}
[\![vpat\ \text{‘}\texttt{+>}\text{’}\ updates]\!]^{\mathsf{A}, st}_{Action} = &\ \text{‘}\texttt{\$(adaptiveSV}\text{’}\ \text{‘}\texttt{[p| \_ |]}\text{’}\ \text{‘}\texttt{[p|}\text{’}\ vpat\ \text{‘}\texttt{|])}\text{’} \\
&\ \text{‘}\texttt{(\textbackslash\_ \_ ->}\text{’}\ \textsc{defaultExpr}(\textsc{ersVars}(\text{‘}\texttt{(}\text{’}\ st\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’}))
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{field}(nt)_{Nonterminal} = nt \\
&\textsc{field}(t)_{Terminal} = \text{‘}\texttt{String}\text{’} \\
&\textsc{field}(p)_{Primitive} = \text{‘}\texttt{(}\text{’}\ p\ \text{‘}\texttt{, String)}\text{’}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{ersVars}(\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ uc\ \text{‘}\texttt{]}\text{’})_{Update} = uc \\
&\textsc{ersVars}(\text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’})_{Update} = \text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ \langle \textsc{ersVars}(u) \mid u \in updates \rangle\ \text{‘}\texttt{)}\text{’} \\
&\textsc{ersVars}(symbol)_{Update} = symbol
\end{aligned}
$$

$$
\begin{aligned}
\textsc{srcCond}(\text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ uconds\ \text{‘}\texttt{)}\text{’})_{UpdateCondition} = &\ \text{‘}\texttt{(}\text{’}\ \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds \rangle) \\
&\ \langle \textsc{srcCond}(uc) \mid uc \in uconds \rangle\ \text{‘}\texttt{)}\text{’}
\end{aligned}
$$

$$
\textsc{srcCond}(symbol)_{UpdateCondition} = \text{‘}\texttt{\_}\text{’}
$$

$$
\begin{aligned}
&\textsc{condHead}(\text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ \ldots\ \text{‘}\texttt{)}\text{’})_{UpdateCondition} = nt \\
&\textsc{condHead}(symbol)_{UpdateCondition} = symbol
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{srcPat}(\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ uc_{Primitive}\ \text{‘}\texttt{]}\text{’})_{Update} = \text{‘}\texttt{(}\text{’}\ var\ \text{‘}\texttt{, \_)}\text{’} \\
&\textsc{srcPat}(\text{‘}\texttt{[}\text{’}\ var\ \text{‘}\texttt{+>}\text{’}\ uc_{Nonterminal}\ \text{‘}\texttt{]}\text{’})_{Update} = var \\
&\textsc{srcPat}(\text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ updates\ \text{‘}\texttt{)}\text{’})_{Update} = \text{‘}\texttt{(}\text{’}\ \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in \textsc{ersVars}(updates) \rangle) \\
&\quad\quad \langle \textsc{srcPat}(u) \mid u \in updates \rangle\ \text{‘}\texttt{)}\text{’} \\
&\textsc{srcPat}(symbol)_{Symbol} = \text{‘}\texttt{\_}\text{’}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{defaultExpr}(symbol)_{Primitive} = \text{‘}\texttt{(undefined, " ")}\text{’} \\
&\textsc{defaultExpr}(symbol)_{Nonterminal} = \textsc{nullCon}(symbol) \\
&\textsc{defaultExpr}(symbol)_{Terminal} = \text{‘}\texttt{" "}\text{’} \\
&\textsc{defaultExpr}(\text{‘}\texttt{(}\text{’}\ nt\ \text{‘}\texttt{->}\text{’}\ uconds\ \text{‘}\texttt{)}\text{’})_{UpdateCondition} = \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds \rangle) \\
&\quad\quad \langle \textsc{defaultExpr}(uc) \mid uc \in uconds \rangle
\end{aligned}
$$

Fig. 5 Semantics of BiYacc programs (as BiGUL programs)

$$
\mathsf{Sub}\ (\mathsf{Num}\ 0)\ \mathsf{y} {+\!\!>}\ \text{‘}-\text{’}\ (\mathsf{y} {+\!\!>}\ \mathsf{Factor})
$$

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

$$\mathsf{entrance} = \mathsf{bigulArithExpr}$$

which is invoked in the main program.

Well-behavedness Since BiGUL programs always denote well-behaved lenses, a fact which has been formally verified [39], we get the following theorem for free.

Theorem 2 (*Well-behavedness*) *The BiGUL program generated from a BiYacc program is a lens, that is, it satisfies the well-behavedness laws in Definition 1 with cst substituted for the source s and ast for the view v*:

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

Handling Grammatical Ambiguity

In Sect. 4, we have described the basic version of BiYacc, about which there is an important assumption (stated in Theorem 1) that grammars have to be unambiguous. Having this assumption can be rather inconvenient in practice, however, as

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

```
#Concrete
Expr -> [Plus]       Expr '+' Expr
      | [Minus]      Expr '-' Expr
      | [Times]      Expr '*' Expr
      | [Division]   Expr '/' Expr
      | [Paren]      '(' Expr ')'
      | [Lit]        Numeric
      ;
```

```
#Directives
Priority:
Times > Plus     ;
Times > Minus    ;
Division > Plus  ;
Division > Minus ;

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] ')'         ;
;;
```

Fig. 6 Arithmetic expressions defined by an ambiguous grammar and the corresponding printing actions. (For simplicity, the variable and negation productions are omitted)

ambiguous grammars (with disambiguation directives) are often preferred since they are considered more natural and human friendly than their unambiguous versions [2, 26]. 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: tools designed for building parser and printer pairs usually do not support such functionality (Sect. 7.1).

For example, consider the ambiguous grammar (with disambiguation directives) and printing actions in Fig. 6, 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 data types and constructors as in Sect. 4.2.1, 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 Fig. 2 is less intuitive as it uses different nonterminals to resolve the ambiguity regarding operator precedence and associativity.

In this section, we explain the problem brought by ambiguous grammars (Sect. 5.1) and address it (Sect. 5.2) using generalised parsing and bidirectionalised filters (bi-filters for short). Then we extend BiYacc with bi-filters (Sect. 5.3) while still retaining the well-behavedness. To program with bi-filters easily, we provide compositional bi-filter directives (Sect. 5.4) which compile to priority and associativity bi-filters. Power users can also define their own bi-filters (Sect. 5.5), and we illustrate this by writing a bi-filter that solves the (in)famous dangling-else problem.

Problems with Ambiguous Grammars

Consider the original architecture of BiYacc in Fig. 3, which we want to (and basically will) retain while adapting it to support ambiguous grammars. The first

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 [23] were invented to describe how to make this choice. For example, with respect to the grammar in Fig. 6, text 1 + 2 * 3 will have either of the two CSTs[4]:

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

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 cst_1.

However, merely adapting *cparse* with disambiguation behaviour is not enough, since the isomorphism (Theorem 1), 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, cparse (cprint cst_2) = Just cst_1 ≠ Just cst_2. This is because the image of *cparse* is strictly smaller than the domain of *cprint*: 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 Sect. 5.3).

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 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 creating a CST with proper parentheses, like $\mathsf{cst}_3 = {}^{\sharp}$ 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 [3, 23], and therefore it is nontrivial to invent a dual semantics in the *put* direction. To have a simple and clear semantics of

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

the disambiguation process, we turn away from YACC's traditional approach and opt for an alternative approach based on generalised parsing with disambiguation filters [9, 26], 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.

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) [14, 47, 50, 54], and works naturally with ambiguous grammars. In practice, a generalised parser can be generated using, e.g., HAPPY's GLR mode [33], and we will assume that given a grammar we can obtain a generalised parser:

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

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 $judge :: \mathsf{Tree} \to \mathsf{Bool}$; 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).[5] For example, to state that top-level addition is left-associative, we can use the following filter[6] 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

[5] 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 the simplified type $t \to \mathsf{Bool}$. See Sect. 7.2.

[6] 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 9) so that ambiguity is fully removed from the grammar.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | ☐ LE | ☐ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 .
```

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

Definition 6 (Bidirectionalised filters) A bidirectionalised filter F working on trees of type t is a function of type $\mathsf{BiFilter}\,t$ defined by:

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

satisfying

$$repair\ F\ t = t' \quad \Rightarrow \quad judge\ F\ t' = \mathsf{True} \qquad \text{(RepairJudge)}$$

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

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 transform its input tree into a state accepted by $judge_F$. 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 2 (***JudgeRepair***) *Any bi-filter F satisfies the JudgeRepair property*:

$$judge_F\, t = \mathsf{True} \quad \Rightarrow \quad repair_F\, t = t.$$

Proof From $judge_F\, t = \mathsf{True}$ we deduce $F\, t = \mathsf{Nothing}$, which implies $repair_F\, t = t$. □

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 between CSTs and ASTs, we require a further property characterising the interaction between the repairing direction of a bi-filter and the get direction of a lens.

Definition 7 (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 (since 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.

The New BiYacc System for Ambiguous Grammars

As depicted in Fig. 7, 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 Fig. 8 and will use this one instead. In the new system, we will still use the *get* and *put* transformations generated from printing actions and 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 Fig. 8) and $repair_F$ components are integrated into the isomorphism and lens parts

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

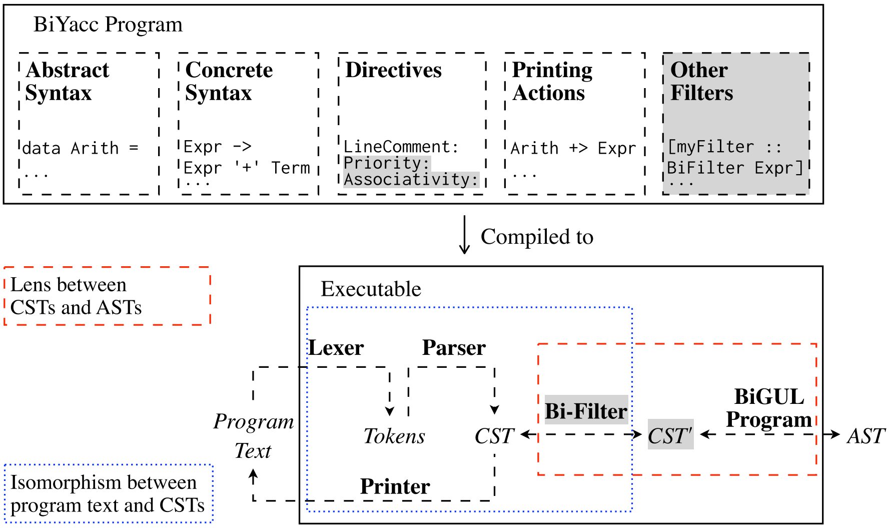

Fig. 7 New architecture of BiYacc (new components are in light grey)

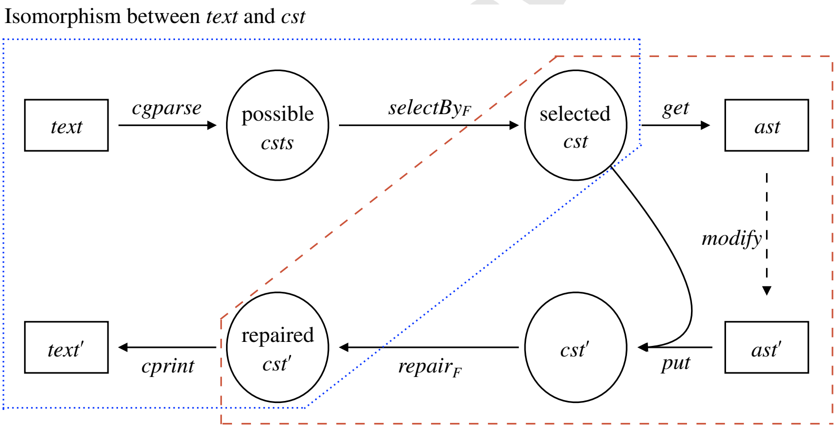

Fig. 8 A schematic diagram showing how parsing and printing work with a bi-filter

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.

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 Fig. 8 and consists of *cprint*, *cgparse*, and $selectBy_F$:

$$
\begin{aligned}
& cprint :: \mathsf{CST} \rightarrow \mathsf{String} \\
& cgparse :: \mathsf{String} \rightarrow [\mathsf{CST}] \\
& selectBy_F :: [\mathsf{CST}] \rightarrow \mathsf{Maybe}\ \mathsf{CST}_F \\
& selectBy_F\ csts = \mathbf{case}\ selectBy\ judge_F\ csts\ \mathbf{of} \\
& \qquad\qquad\qquad\qquad [cst] \ \rightarrow \mathsf{Just}\ cst \\
& \qquad\qquad\qquad\qquad \_ \qquad \rightarrow \mathsf{Nothing} \\
& selectBy :: (a \rightarrow \mathsf{Bool}) \rightarrow [a] \rightarrow [a] \\
& selectBy\ p[\,] = [\,] \\
& selectBy\ p(x : xs) | px = x : selectBy\ p\ xs \\
& selectBy\ p(x : xs) | otherwise = selectBy\ p\ xs\ .
\end{aligned}
$$

In the parsing direction, first *cgparse* produces all the CSTs; then $selectBy_F$ 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, $\mathsf{Nothing}$ 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:

$$
\begin{aligned}
cparse_F &:: \mathsf{String} \rightarrow \mathsf{Maybe}\ \mathsf{CST}_F \\
cparse_F &= selectBy_F \circ cgparse \\
cprint_F &:: \mathsf{CST}_F \rightarrow \mathsf{Maybe}\ \mathsf{String} \\
cprint_F &= \mathsf{Just} \circ cprint\ .
\end{aligned}
$$

We are eager to give the revised version of the inverse properties (Theorem 3) and their proofs, which, however, depend on two assumptions about generalised parsers and bi-filters. So let us present them in order.

Definition 8 (Generalised parser correctness) A generalised parser *cgparse* is correct with respect to a printer *cprint* exactly when

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

This is exactly Definition 3.7 of Klint and Visser [26]. We remind the reader again that we use sets and lists interchangeably for the parsing results.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

Definition 9 (Bi-filter completeness) A bi-filter F is complete with respect to a printer *cprint* exactly when

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

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

This is revised from Definition 4.3 of Klint and Visser [26], 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 [10], we cannot tell whether a (bi-)filter (for the full CFG) is complete, either. But still, some checks can be performed in simple cases, as stated in Sect. 7.

The following two lemmas connect our two assumptions, Definitions 8 and 9, with the definitions of cparse_F and cprint_F.

Lemma 3 *Given cparse_F and cprint_F where cgparse is correct and F is complete with respect to cprint, we have*

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

Proof We reason:

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

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

Lemma 4 (*Printer injectivity*) *If F is a complete bi-filter, then cprint_F is injective.*

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

We can now prove a generalised version of Theorem 1 for ambiguous grammars.

Theorem 3 *(Inverse properties with bi-Filters) Given $cparse_F$ and $cprint_F$ where cgparse is correct and F is complete, we have the following:*

$$cparse_F\ text = \mathsf{Just}\ cst \quad \Rightarrow \quad cprint_F\ cst \ = \mathsf{Just}\ text \tag{6}$$

$$cprint_F\ cst = \mathsf{Just}\ text \quad \Rightarrow \quad cparse_F\ text = \mathsf{Just}\ cst\ . \tag{7}$$

Proof For (6): 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 (7): the antecedent implies $cprint\ cst = text$. By Lemma 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 we know $cst' = cst$, and thus $cparse_F\ text = \mathsf{Just}\, cst$. □

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:

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

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:

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

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

$$\begin{aligned} fmap\ &::\ (a \rightarrow b) \rightarrow \mathsf{Maybe}\, a \rightarrow \mathsf{Maybe}\, b \\ fmapf\ \mathsf{Nothing}\ &=\ \mathsf{Nothing} \\ fmapf\ (\mathsf{Just} x)\ &=\ \mathsf{Just}(fx)\ . \end{aligned}$$

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

Lemma 5 *If $fmap\ f\ mx = \mathsf{Just}\ y$, 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 2 for ambiguous grammars.

Theorem 4 (*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*:

$$put_F\ cst\ ast = \mathsf{Just}\ cst' \quad \Rightarrow \quad get_F\ cst' = \mathsf{Just}\ ast \tag{8}$$

$$get_F\ cst = \mathsf{Just}\ ast \quad \Rightarrow \quad put_F\ cst\ ast = \mathsf{Just}\ cst\ . \tag{9}$$

Proof For (8): the antecedent expands to $fmap\ repair_F\ (put\ cst\ ast) = \mathsf{Just}\ cst'$, which, by Lemma 5, implies $put\ cst\ ast = \mathsf{Just}\ cst''$ for some cst'' such that $repair_F\ cst'' = cst'$. Now we reason:

$$\begin{aligned} & get_F\ cst' \\ = & \quad \{\ \text{Definition of } get_F \text{ and } cst \in \mathsf{CST}_F \} \\ & get\ cst' \\ = & \quad \{\ \text{Definition of } cst'\ \} \\ & get\ (repair_F\ cst'') \\ = & \quad \{\ \text{PassThrough}\ \} \\ & get\ cst'' \\ = & \quad \{\ \text{PutGet}\ \} \\ & \mathsf{Just}\ ast\ . \end{aligned}$$

For (9):

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

□

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 (Sect. 5.4.1) so that we can build large bi-filters from small ones. This is a

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

suitable semantic foundation for introducing YACC-like directives for specifying priority and associativity into BIYACC (Sect. 5.4.2), 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 (Sect. 5.4.3).

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 6, 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 10 (Bi-filter composition) The composition of two bi-filters is defined by

$$
\begin{aligned}
&(\lhd) :: (t \to \mathsf{Maybe}\ t) \to (t \to \mathsf{Maybe}\ t) \to (t \to \mathsf{Maybe}\ t)\\
&(j \lhd i)\, t = \mathbf{case}\ i\ t\ \mathbf{of}\\
&\qquad\qquad \mathsf{Nothing} \to j\, t\\
&\qquad\qquad \mathsf{Just}\ t' \quad \to \mathbf{case}\ j\ t'\ \mathbf{of}\\
&\qquad\qquad \mathsf{Nothing} \to \mathsf{Just}\ t'\\
&\qquad\qquad \mathsf{Just}\ t'' \quad \to \mathsf{Just}\ t''\ .
\end{aligned}
$$

When applying a composite bi-filter $j \lhd i$ 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 $\mathsf{Nothing}$.

Lemma 6 *For a composite bi-filter $j \lhd i$, the following two equations hold*:

$$
\begin{aligned}
repair\,(j \lhd i)\, t &= (repair_j \circ repair_i)\, t\\
judge(j \lhd i)\, t &= judge_j\, t \wedge judge_i\, t\ .
\end{aligned}
$$

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

$$
judge_i\, t = \mathsf{True} \quad \Rightarrow \quad judge_i\,(repair_j\, t) = \mathsf{True}\ .
$$

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 j after i. This is proved as the following theorem.

Theorem 5 *Let i and j be bi-filters (satisfying RepairJudge and PassThrough). If j respects i, then $j \lhd i$ also satisfy RepairJudge and PassThrough.*

Proof For RepairJudge, we reason:

$$
\begin{aligned}
& judge(j \lhd i)(repair\ (j \lhd i)\ t) \\
=\ & \{\ \text{Lemma 6}\ \} \\
& judge(j \lhd i)(repair_j(repair_i t)) \\
=\ & \{\ \text{Lemma 6}\ \} \\
& judge_j(repair_j(repair_i t)) \wedge judge_i(repair_j(repair_i t)) \\
=\ & \{\ \text{RepairJudge of } j\ \} \\
& \mathsf{True} \wedge judge_i(repair_j(repair_i t)) \\
=\ & \{ judge_i(repair_i t') = \mathsf{True};\ j \text{ respects } i\ \} \\
& \mathsf{True} \wedge \mathsf{True} \\
=\ & \mathsf{True}\ .
\end{aligned}
$$

And for PassThrough:

$$
\begin{aligned}
& get(repair(j \lhd i) t) \\
=\ & \{\ \text{Lemma 6}\ \} \\
& get(repair_j\ (repair_i t)) \\
=\ & \{ \text{PassThrough of } j\ \} \\
& get(repair_i t) \\
=\ & \{ \text{PassThrough of } i\ \} \\
& gett\ .
\end{aligned}
$$

□

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[7] 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 [24]

[7] The YACC-style approach adopts the word *precedence* [23] while the filter-based approaches tend to use the word priority [9, 26]. We follow the traditions and use either word depending on the context.

Springer Ohmsha
Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

(chapter *Expressions*) and HASKELL [34] (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 (%left, %right, and %nonassoc) 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.

(The bi-filter semantics of) our bi-filter directives repair CSTs violating priority and associativity constraints by adding parentheses—for example, if the production of addition expressions in Fig. 6 is left-associative, then we can repair ♯ Plus 1 (Plus 2 3) by adding parentheses around the right subtree, yielding ♯ Plus 1 (Paren (Plus 2 3)), provided that the grammar has a production of parentheses annotated with the bracket attribute [8, 53]:

```
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 data type 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 Sect. 5.5).

In order for bi-filter directives to work correctly, the user should notice the following requirements: (1) directives shall not mention the parenthesis production annotated with bracket attribute so that they respect each other and work properly (as introduced in Definition 11). (2) Suppose that the parenthesis production is $NT \rightarrow \alpha NT_R \beta$ where α and β denote a sequence of terminals and NT_R is a possibly different nonterminal from NT (on the right-hand side of the production)—for instance, Expr -> '(' Expr ')' above— there shall be exactly one printing action defined for the parenthesis production in the form of v +> $\alpha[v$ +> $NT_R]\beta$ for the PassThrough property to hold: for any CST, the (added) parenthesis structure will all be dropped through the conversion to its AST.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 Sect. 5.4.3.

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 lower priority is a direct child of the node of higher priority. For instance, we can define that (the production of) multiplication has higher priority than (the production of) addition for the grammar in Fig. 6 by writing

$$\mathsf{Expr} \mathrel{{-}{>}} \mathsf{Expr}\ `{*}\text{'}\ \mathsf{Expr} \;>\; \mathsf{Expr} \mathrel{{-}{>}} \mathsf{Expr}\ `{+}\text{'}\ \mathrm{Expr}\ ;$$
$$\text{or just}\quad \mathsf{Times} > \mathsf{Plus};\ .$$

The directive first produces the following top-level bi-filter:[8]

```
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 $or :: [\mathsf{Bool}] \rightarrow \mathsf{Bool}$. If there is any incorrect part, we repair it by inserting a parenthesis structure using addPar.

In general, the syntax of priority directives is

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

where *Constructor* and *Symbol* are already defined in Fig. 4; for each priority declaration, we can use either productions or their names (i.e. constructors).

[8] Although terminals such as '*' and '+' are uniquely determined by constructors and not explicitly included in the CSTs, there are fields in CSTs for holding whitespaces 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.

Springer Ohmsha

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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

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

$\textsc{con}$ looks up constructor names for input productions (divided into nonterminals and right-hand sides); $\textsc{fillVars}(nt)$ generates variable names for each terminal and nonterminal in nt (here RHS_1); $\textsc{fillUndefined}$ is similar to $\textsc{fillVars}$ 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 Fig. 6) 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:

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

```
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 $^\sharp$`Plus (Plus 1 2) 3` but not $^\sharp$`Plus 1 (Plus 2 3)`, which is repaired to $^\sharp$`Plus 1 (Paren (Plus 2 3))`.

Generally, the syntax of associativity directives is

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

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 $NT_1 \rightarrow \alpha_1 NT_{1R}$ and $NT_2 \rightarrow \alpha_2 NT_{2R}$ 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

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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

Functions CON, FILLUNDEFINED, and FILLVAR have the same behaviour as before; FILLVARS−FROM (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) [9], 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 $\texttt{cst}_1 = {}^\sharp\texttt{Plus (FromE 1)}$ and $\texttt{cst}_2 = {}^\sharp\texttt{Times (FromE (Plus 1 2) (FromE 3))}$, and we want to spot `(FromE (Times 2 3))` 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).

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* [30]), 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 matching 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 Sect. 5.4.2), 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 matching 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 6, we should write t+> '(' [t+> Expr] ')' as the only printing action mentioning parentheses, which means that $put\ (\mathsf{Paren}\ s)\ t = fmap\ \mathsf{Paren}\ (put\ s\ t)$ for all s and t. Then the following reasoning implies that $get\ (\mathsf{Paren}\ s) = get\ s$ for all s:

$$
\begin{aligned}
& get\ (\mathsf{Paren}\ s) = \mathsf{Just}\ t \\
\Leftrightarrow\ & \{\Rightarrow \text{ by GetPut and } \Leftarrow \text{ by PutGet}\} \\
& put\ (\mathsf{Paren}\ s)\ t = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow\ & \{\text{By the above statement: } put\ (\mathsf{Paren}\ s)\ t = fmap\ \mathsf{Paren}\ (put\ s\ t)\} \\
& fmap\ \mathsf{Paren}\ (put\ s\ t) = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow\ & \{\text{Lemma 5 and the definition of } fmap\} \\
& put\ s\ t = \mathsf{Just}\ s \\
\Leftrightarrow\ & \{\Rightarrow \text{ by PutGet and } \Leftarrow \text{ by GetPut}\} \\
& get\ s = \mathsf{Just}\ t
\end{aligned}
$$

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 $i \lhd j$ and $j \lhd i$ may have different behaviours, 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.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

Definition 12 (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 6, this implies $repair\ (i \lhd j) = repair\ (j \lhd i)$. Note that $judge\ (i \lhd j) = judge\ (j \lhd i)$ 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 7 *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: (1) repairing does not introduce more prohibited pairs of productions, and (2) 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 (1) 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 (2) 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.

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 Fig. 4. The syntax is

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

$$\mathit{OtherFilters} ::= \text{‘[’}\ \mathit{HsFunDecl}^{+}\{\text{‘,’}\}\ \text{‘]’}\ \mathit{HsCode}$$
$$\mathit{HsFunDecl} ::= \mathit{HsFunName}\ \text{‘}\ \texttt{:: BiFilter}\ \text{’}\ \mathit{Nonterminal}\ .$$

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 [26], 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 [26].

Klint and Visser [26] 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 [26]. 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, since it (1) only searches for non-parenthesis productions that are not declared in any other directives, and (2) inserts only a parenthesis production when repairing

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

incorrect CSTs. The reader may find the code of checkRightSpine in more detail in Fig. 10.

Case Studies

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. For example, Kinoshita and Nakano [25] adopted BiYacc as part of their system for synchronising Coq functions and corresponding OCaml programs. In this section, we demonstrate BiYacc with a medium-size case study: we use BiYacc to build a pair of parser and reflective printer for the Tiger language [4] and demonstrate some of their uses.

The Tiger Language

Tiger is a statically typed imperative language first introduced in Appel's textbook on compiler construction [4]. 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 reflective 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 without any disambiguation declarations (based on Kernighan and Ritchie [24] 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 Fig. 9. 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 [21]. Concretely, we add a parenthesis production to the grammar (and discard it when converting CSTs to ASTs, so that the PassThrough property could be satisfied), since 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. There is also another slight change in the definition

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

```
#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 ;

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 ;
...
```

Fig. 9 An excerpt of TIGER's abstract and concrete syntax. (Here we define our own BBool type and MMaybe type to avoid name clashing with HASKELL's built-in ones)

of ASTs for handling a feature not supported by the current BIYACC: the AST constructors TFunctionDec and TTypeDec take a single function or type declaration instead of a list of adjacent declarations (for representing mutual recursion) as in Appel [4], since we cannot handle the synchronisation between a list of lists (in ASTs) and a list (in CSTs) with BIYACC's current syntax.

Following Hirzel and Rose's specification [21], the disambiguation directives for TIGER are shown in Fig. 10; for instance, we define multiplication to be left-associative. The directives also include a concrete treatment for the dangling else problem, which is usually 'not solved' when using a YACC-like (LA)LR parser generator to implement parsers: rather than resolving the grammatical ambiguity, we often rely on the default behaviour of the parser generator—preferring shift.

Journal : Small-ext 354	Dispatch : 13-2-2020	Pages : 55
Article No. : 82	□ LE	□ TYPESET
MS Code : NGCO-D-19-00023R2	☑ CP	☑ DISK

```
#Directives                           Associativity:
Priority:                             Left:  Times, Plus, And  ... ;
Times >  Plus ;                       Right: Assign,  ... ;
And   >  Or   ; ...

#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"]
...
```

Fig. 10 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)

We have successfully tested our BIYACC program for TIGER on all the sample programs provided on the homepage of Appel's book,[9] 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 reflective printing, can achieve.

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 (usually 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:

9FL01 [9] https://www.cs.princeton.edu/~appel/modern/testcases/.

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

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 reflective printer can make the correct decision by inspecting the CST.

The idea of resugaring [42] 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 [42, 43].

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 6.2. For the 'not' operator, which Tiger lacks, we introduce '$\sim$', 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 reflective 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 PutGet 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 an enriched abstract syntax.

Language Evolution

When a language evolves, some new features of the language (e.g. the foreach loops introduced in Java 5 [20]) 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 [18] 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 Dijkstra's guarded command language [12]. We extend TIGER's concrete syntax with the following production rules:

```
Exp    -> ... | Guard | ... ;         CaseBs -> CaseB CaseBs | CaseB ;
Guard  -> 'guard' CaseBs 'end';       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 reflective 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 being 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

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | LE | TYPESET
MS Code : NGCO-D-19-00023R2 | CP | DISK

```
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 reflected 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 .
```

Reflective 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 completely disrupt the matching between the AST and the CST.

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 (1) simplification-like optimisations such as constant folding and constant propagation and (2) 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.

Related Work

Unifying Parsing and Printing

Much research has been devoted to describing parsers and printers in a single program. For example, both Rendel and Ostermann [44] and Matsuda and Wang [36, 37] adopt a combinator-based approach[10] (whereas we use a generator-based approach), where small components are glued together to yield more sophisticated behaviour, and can guarantee properties similar to Theorem 1 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 [38] to realise the parsing semantics. Brabrand et al. [7] 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 reflective 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 basic BiYacc (that 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)

[10] Although they use different implementation techniques, we will not dive into them in our related work. See Matsuda and Wang's related work for a comparison [36].

Journal : Small-ext 354 | Dispatch : 13-2-2020 | Pages : 55
Article No. : 82 | □ LE | □ TYPESET
MS Code : NGCO-D-19-00023R2 | ☑ CP | ☑ DISK

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 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. [7] 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 [10].

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 [9, 26, 52, 53], and thus the completeness for simple (bi-)filters (see Definition 9) 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 Sect. 5.4.2, 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 BIYACC with an industrial tool, AUGEAS, which provides the user with a local configuration API that converts configuration data into a rose tree representation [31]. 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: (1) AUGEAS works for regular grammars while BIYACC works for (unambiguous) context-free grammars. (2) AUGEAS uses a combinator-based approach while BIYACC adopts a generator-based approach. (3) 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 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 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).

Generalised Parsing, Disambiguation, and Filters

The grammar of a programming language is usually designed to be unambiguous. Various parser-dependent disambiguation methods such as grammar transformation [29] and parse table conflicts elimination [23] have been developed to guide the parser to produce a single correct CST [26]. 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 [14] and generalised LR [50] (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' [48].

The idea to relate generalised parsing with parser-independent disambiguation for programming languages is proposed by Klint and Visser [26]. They proposed two classes of filters, property filters (defined in terms of predicates on a single tree) and comparison filters (defined in terms of relations among trees), but we only adapt and bidirectionalise predicate filters in this paper. One difficulty lies in the fact that it is unclear how to define *repair* 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 [29] and Aasa's [1] 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[11] (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 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). 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 [8] 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 [8] 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.)

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 [23] 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 [29], 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

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

impose bi-filter laws, which further makes it possible to guarantee the well-behavedness of the whole system.

Comparison with a Get-Based Approach

Our work is theoretically based on asymmetric lenses [17] of bidirectional transformations [11, 19], particularly taking inspiration from the recent progress on putback-based bidirectional programming [15, 27, 28, 40, 41]. As explained in Sect. 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. We call a language get-based when programs written in the language denote *get* functions, and call a language putback-based when its programs denote *put* functions. In the context of parsing and reflecting printing, the get-based approach lets the programmer describe a parser, whereas the putback-based approach lets the programmer describe a printer. Below we discuss in more depth how the putback-based methodology affects BiYacc's design by comparing BiYacc with a closely related, get-based system.

Martins et al. [35] 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 Fig. 2 can be expressed in Martins et al.'s system as

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

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 putback-based approaches differ in the context of parsing and reflective 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 putback-based system can give the user fuller control. For example, when no source is given and more than one rule 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. For example: suppose that we port to Martins et al.'s system the BiYacc action that relates Tiger's concrete '&' operator with a specialised abstract `if` expression in Sect. 6.2, coexisting with a more general rule that maps a concrete `if` expression to an abstract `if` expression. Then printing the AST `TCond (TSV "a") (TSV "b") 0` from scratch will and can only produce `a & b`, 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 & b` or `if a then b else 0` by suitably ordering the printing actions.

Journal : Small-ext 354 Dispatch : 13-2-2020 Pages : 55
Article No. : 82 □ LE □ TYPESET
MS Code : NGCO-D-19-00023R2 ☑ CP ☑ DISK

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 putback-based design. The user of Martins et al.'s system writes rules to specify a forward 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 putback-based thinking.

Conclusion

We conclude the paper by summarising our contributions:

- We have presented the design and implementation of BiYacc, with which the programmer can describe both a parser and a reflective printer for a fully disambiguated context-free grammar in a single program. Our solution guarantees the partial version of the consistency properties (Definition 2) by construction.
- We proposed the notion of bi-filters, which enables BiYacc to disambiguate ambiguous grammars while still respecting the consistency properties. This is the main new contribution compared to the previous SLE'16 version [55].
- We have demonstrated that BiYacc can support various tasks of language engineering, from traditional constructions of basic machinery such as printers and parsers to more complex tasks such as resugaring, simple refactoring, and language evolution.

Acknowledgements We thank the reviewers and the editor for their selflessness and effort spent on reviewing our paper, a quite long one. With their help, the readability of the paper is much improved, especially regarding how several case studies are structured, how theorems for the basic BiYacc and theorems for the extended version handling ambiguous grammars are related, and how look-alike notions are 'disambiguated'. This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099; in particular, most of the second author's contributions were made when he worked at the National Institute of Informatics and funded by the Grant.

References

1. Aasa, A.: Precedences in specifications and implementations of programming languages. In: Selected Papers of the Symposium on Programming Language Implementation and Logic Programming, Elsevier Science Publishers B. V., Amsterdam, PLILP '91, pp. 3–26. http://dl.acm.org/citation.cfm?id=203429.203431 (1995)

Journal : Small-ext 354	Dispatch : 13-2-2020	Pages : 55
Article No. : 82	□ LE	□ TYPESET
MS Code : NGCO-D-19-00023R2	☑ CP	☑ DISK

2. Afroozeh, A., Izmaylova, A.: Faster, practical GLL parsing. In: Franke, B. (ed.) Compiler Construction, pp. 89–108. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46663-6_5
3. Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic parsing of ambiguous grammars. Commun. ACM **18**(8), 441–452 (1975)
4. Appel, A.W.: Modern Compiler Implementation in ML, 1st edn. Cambridge University Press, New York (1998)
5. Bird, R.: Thinking Functionally with Haskell. Cambridge University Press, Cambridge (2014). https://doi.org/10.1017/CBO9781316092415
6. Boulton, R.: Syn: a single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Tech. Rep. Number 390, Computer Laboratory, University of Cambridge (1966)
7. Brabrand, C., Møller, A., Schwartzbach, M.I.: Dual syntax for XML languages. Inf. Syst. **33**(4–5), 385–406 (2008). https://doi.org/10.1016/j.is.2008.01.006
8. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Trans. Softw. Eng. Methodol. **5**(1), 1–41 (1996). https://doi.org/10.1145/226155.226156
9. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Proceedings of the 11th International Conference on Compiler Construction, Springer, London, UK, CC '02, pp. 143–158. https://doi.org/10.1007/3-540-45937-5_12 (2002)
10. Cantor, D.G.: On the ambiguity problem of Backus systems. J. ACM **9**(4), 477–479 (1962)
11. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, Springer, Berlin, ICMT '09, pp. 260–283. https://doi.org/10.1007/978-3-642-02408-5_19 (2009)
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
13. Duregård, J., Jansson, P.: Embedded parser generators. In: Proceedings of the 4th ACM Symposium on Haskell, ACM, New York, NY, USA, Haskell '11, pp. 107–117. https://doi.org/10.1145/2034675.2034689 (2011)
14. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2), 94–102 (1970). https://doi.org/10.1145/362007.362035
15. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. Sci. China Inf. Sci. **58**(5), 1–21 (2015)
16. Foster, J.N.: Bidirectional programming languages. PhD thesis, University of Pennsylvania (2009)
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**, 3 (2007). https://doi.org/10.1145/1232420.1232424
18. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)
19. Gibbons, J., Stevens, P.: International Summer School on Bidirectional Transformations (Oxford, UK, 25–29 July 2016). Lecture Notes in Computer Science, vol. 9715. Springer, Berlin (2018)
20. Gosling, J., Joy, B., Steele, G.: The Java Language Specification, 3rd ed (2006). https://docs.oracle.com/javase/specs/
21. Hirzel, M., Rose, K.H.: Tiger language specification (2013). https://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001/tiger-spec.pdf
22. Hu, Z., Ko, H.S.: Principles and practice of bidirectional programming in BiGUL. In: Gibbons, J., Stevens, P. (eds.) Bidirectional Transformations: International Summer School, Oxford, UK, July 25–29, 2016, Tutorial Lectures, pp. 100–150. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79108-1_4
23. Johnson, S.C.: Yacc: Yet another compiler-compiler. AT&T Bell Laboratories Technical Reports (AT&T Bell Laboratories Murray Hill, New Jersey 07974). p. 32 (1975)
24. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall Press, Upper Saddle River (1988)
25. Kinoshita, D., Nakano, K.: Bidirectional certified programming. In: Eramo, R., Johnson. M. (eds) Proceedings of the 6th International Workshop on Bidirectional Transformations Co-Located with The European Joint Conferences on Theory and Practice of Software (ETAPS 2017), CEUR Workshop Proceedings, Uppsala, Sweden, vol. 1827, pp. 31–38 (2017)

Journal : **Small-ext 354**	Dispatch : **13-2-2020**	Pages : **55**
Article No. : **82**	□ LE	□ TYPESET
MS Code : **NGCO-D-19-00023R2**	☑ CP	☑ DISK

26. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Pighizzini, G., Pietro, P.S. (eds) Proceedings of the ASMICS Workshop on Parsing Theory, University of Milan, Italy, Milano, Italy, pp. 1–20 (1994)
27. Ko, H.S., Hu, Z.: An axiomatic basis for bidirectional programming. Proc. ACM Program. Lang. **2**(POPL), 41:1–41:29 (2018). https://doi.org/10.1145/3158129
28. Ko, H.S., Zan, T., Hu, Z.: 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, ACM, New York, NY, USA, PEPM '16, pp. 61–72 (2016). https://doi.org/10.1145/2847538.2847544
29. LaLonde, W.R., des Rivieres, J.: Handling operator precedence in arithmetic expressions with tree transformations. ACM Trans. Program. Lang. Syst. **3**(1), 83–103 (1981). https://doi.org/10.1145/357121.357127
30. Lämmel, R., Jones, S.P.: 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, ACM, New York, NY, USA, TLDI '03, pp. 26–37 (2003). https://doi.org/10.1145/604174.604179
31. Lutterkort, D.: Augeas—a configuration API. In: Proceedings of the Ottawa Linux Symposium, Ottawa, Canada, pp. 47–56 (2008)
32. Macedo, N., Pacheco, H., Cunha, A., Oliveira, J.N.: Composing least-change lenses. Proc. Sec. Int. Workshop Bidirect. Transform. **57**, 1–19 (2013). https://doi.org/10.14279/tuj.eceasst.57.868
33. Marlow, S., Gill, A.: The parser generator for Haskell. https://www.haskell.org/happy/ (2001)
34. Marlow, S., et al.: Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/ (2010)
35. Martins, P., Saraiva, J., Fernandes, J.P., Van Wyk, E.: Generating attribute grammar-based bidirectional transformations from rewrite rules. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, ACM, New York, NY, USA, PEPM '14, pp. 63–70 (2014). https://doi.org/10.1145/2543728.2543745
36. Matsuda, K., Wang, M.: Embedding invertible languages with binders: a case of the FliPpr language. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, ACM, New York, NY, USA, Haskell 2018, pp. 158–171 (2018a). https://doi.org/10.1145/3242744.3242758
37. Matsuda, K., Wang, M.: FliPpr: a system for deriving parsers from pretty-printers. New Gener. Comput. **36**(3), 173–202 (2018b). https://doi.org/10.1007/s00354-018-0033-7
38. Matsuda, K., Mu, S.C., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: Gordon, A.D. (ed) Proceedings of the 19th European Conference on Programming Languages and Systems, Springer, Berlin, no. 20 in ESOP'10, pp. 448–467 (2010). https://doi.org/10.1007/978-3-642-11957-6_24
39. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
40. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, ACM, New York, NY, USA, PEPM '14, pp. 39–50 (2014a). https://doi.org/10.1145/2543728.2543737
41. Pacheco, H., Zan, T., Hu, Z.: BiFluX: A bidirectional functional update language for XML. In: Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, ACM, New York, NY, USA, PPDP '14, pp. 147–158 (2014b). https://doi.org/10.1145/2643135.2643141
42. Pombrio, J., Krishnamurthi, S.: Resugaring: lifting evaluation sequences through syntactic sugar. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, no. 6 in PLDI '14, pp. 361–371 (2014). https://doi.org/10.1145/2594291.2594319
43. Pombrio, J., Krishnamurthi, S.: Hygienic resugaring of compositional desugaring. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, no. 13 in ICFP 2015, pp. 75–87 (2015). https://doi.org/10.1145/2784731.2784755
44. Rendel, T., Ostermann, K.: Invertible syntax descriptions: unifying parsing and pretty printing. In: Proceedings of the Third ACM Haskell Symposium on Haskell, ACM, New York, NY, USA, Haskell '10, pp. 1–12 (2010). https://doi.org/10.1145/1863523.1863525

Journal : **Small-ext 354** Dispatch : **13-2-2020** Pages : **55**
Article No. : **82** □ LE □ TYPESET
MS Code : **NGCO-D-19-00023R2** ☑ CP ☑ DISK

45. Reps, T., Teitelbaum, T.: The synthesizer generator. In: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, New York, NY, USA, SDE 1, pp. 42–48 (1984). https://doi.org/10.1145/800020.808247
46. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language-based editors. ACM Trans. Program. Lang. Syst. **5**(3), 449–477 (1983). https://doi.org/10.1145/2166.357218
47. Scott, E., Johnstone, A.: GLL parsing. Electron. Notes Theor. Comput. Sci. **253**(7), 177–189 (2010). https://doi.org/10.1016/j.entcs.2010.08.041
48. Scott, E., Johnstone, A., Economopoulos, R.: BRNGLR: a cubic tomita-style GLR parsing algorithm. Acta Inform. **44**(6), 427–461 (2007). https://doi.org/10.1007/s00236-007-0054-z
49. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, ACM, New York, NY, USA, Haskell '02, pp. 1–16 (2002). https://doi.org/10.1145/581690.581691
50. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence-Volume 2, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI'85, pp. 756–764 (1985). http://dl.acm.org/citation.cfm?id=1623611.1623625
51. Traver, V.J.: On compiler error messages: what they say and what they mean. Ad. Hum. Comput. Interact. **2010**, 3:1–3:26 (2010). https://doi.org/10.1155/2010/602570
52. Visser, E.: A case study in optimizing parsing schemata by disambiguation filters. International Workshop on Parsing Technology (IWPT 1997), pp. 210–224. Massachusetts Institute of Technology, Boston, USA (1997a)
53. Visser, E.: Syntax definition for language prototyping. PhD thesis, University of Amsterdam (1997b)
54. Younger, D.H.: Recognition and parsing of context-free languages in time n^3. Inf. Control **10**(2), 189–208 (1967)
55. Zhu, Z., Zhang, Y., Ko, H.S., Martins, P., Saraiva, J., Hu, Z.: Parsing and reflective printing, bidirectionally. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, ACM, New York, NY, USA, SLE 2016, pp. 2–14. https://doi.org/10.1145/2997364.2997369 (2016)

Affiliations

Zirun Zhu[1] · Hsiang-Shang Ko[2] · Yongzhe Zhang[1] · Pedro Martins[3] · João Saraiva[4] · Zhenjiang Hu[5]

✉ Zirun Zhu
zhu@nii.ac.jp

Hsiang-Shang Ko
joshko@iis.sinica.edu.tw

Yongzhe Zhang
zyz915@nii.ac.jp

Pedro Martins
pribeiro@uci.edu

João Saraiva
jas@di.uminho.pt

Zhenjiang Hu
huzj@pku.edu.cn

Journal : **Small-ext 354**	Dispatch : **13-2-2020**	Pages : **55**
Article No. : **82**	☐ LE	☐ TYPESET
MS Code : **NGCO-D-19-00023R2**	☑ CP	☑ DISK

[1] National Institute of Informatics, Chiyoda, Tokyo, Japan
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan
[3] University of California, Irvine, USA
[4] Department of Informatics, University of Minho, Braga, Portugal
[5] Department of Computer Science and Technology, Peking University, Beijing, China

Journal : **Small-ext 354**	Dispatch : **13-2-2020**	Pages : **55**
Article No. : **82**	□ LE	□ TYPESET
MS Code : **NGCO-D-19-00023R2**	☑ CP	☑ DISK