Parsing and Reflective Printing, Bidirectionally

Zirun Zhu
SOKENDAI (The Graduate University for Advanced Studies), Japan
National Institute of Informatics, Japan
zhu@nii.ac.jp

Yongzhe Zhang
SOKENDAI (The Graduate University for Advanced Studies), Japan
National Institute of Informatics, Japan
zyz915@nii.ac.jp

Hsiang-Shang Ko
National Institute of Informatics, Japan
hsiang-shang@nii.ac.jp

Pedro Martins
University of California, Irvine, USA
pribeiro@uci.edu

João Saraiva
University of Minho, Portugal
jas@di.uminho.pt

Zhenjiang Hu
SOKENDAI (The Graduate University for Advanced Studies), Japan
National Institute of Informatics, Japan
hu@nii.ac.jp

Abstract

Language designers usually need to implement parsers and printers. Despite being two intimately 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 one 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 reflect 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 an unambiguous context-free grammar. BIYACC is based on the theory of *bidirectional transformations*, which helps to guarantee by construction that the pairs of parsers and reflective printers generated by BIYACC are consistent. We show that BIYACC is capable of facilitating many tasks such as Pombrio and Krishnamurthi's "resugaring", language evolution, and refactoring.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized Application Languages

Keywords parsing, reflective printing, bidirectional transformations, domain-specific languages

1. 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 intimately 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 (Boulton 1966; Visser 1997; Rendel and Ostermann 2010; Duregård and Jansson 2011; Matsuda and Wang 2013) 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 Figure 1: 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))
```

Figure 1. Comparing conventional and reflective printing

original program text is an arithmetic expression, containing negation, (redundant) parentheses, and a comment. 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. Then 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 reflect the change on the AST back to the program text to make it easy for the user to check where the changes are. With a conventional printer, however, the printed result will 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 programmer to compare the updated and original versions of the text. The problem illustrated here also occurs in many other practical situations where the parser and printer are used as a bridge between the system and the user, for example,

- in program debugging where part of the original program in an error message is displayed much differently from its original form (De Jonge 2002; Kort and Lämmel 2003), and
- in program refactoring where the comments and layouts in the original program are completely lost after the AST is transformed by the system (de Jonge and Visser 2012).

To address the problem, we propose a domain-specific language BiYacc, which lets the user describe both a parser and a *reflective* printer for an unambiguous context-free grammar 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 program text, and reflects the modification 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 as shown in Figure 1. It is worth noting that reflective printing is a generalisation of the conventional notion of printing, because our 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 programs are *consistent* with each other. Specifically, we want to ensure two inverse-like properties: Firstly, a piece of program text s printed from an abstract syntax tree t should be parsed to the same tree t, i.e.,

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

Secondly, updating a piece of program text s with an abstract syntax tree parsed from s should leave s unmodified (including formatting details like parentheses and spaces), i.e.,

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

These two properties are inspired by the theory of *bidirectional transformations* (Czarnecki et al. 2009; Hu et al. 2011), 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 input examples with various test cases used in the paper. The structure of the paper is as follows:

- We first give an overview of BiYacc in Section 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 Section 3, in particular the bidirectional programming language BiGUL (Ko et al. 2016), we give the semantics of BiYacc by compiling it to BiGUL in Section 4, guaranteeing the properties (1) and (2) by construction.
- We present a case study in Section 5, showing that even though BiYacc is currently restricted to dealing with unambiguous grammars, it is capable of describing Tiger (Appel 1998), which shares many similarities with full grown, widely used languages. We demonstrate that BiYacc can handle syntactic sugar, partially subsume Pombrio and Krishnamurthi (2014)'s "resugaring", and facilitate language evolution.
- Related work including detailed comparison with other systems is presented in Section 6 and in Section 7 we conclude.

2. A first look at BiYacc

We first give an overview of BiYacc by going through the BiYacc program shown in Figure 2, which deals with the arithmetic expression example mentioned in Section 1.

```
1  Abstract
2
3  data Arith = Num Int
4             | Var String
5             | Add Arith Arith
6             | Sub Arith Arith
7             | Mul Arith Arith
8             | Div Arith Arith
9    deriving (Show, Eq, Read)
10
11 Concrete
12
13 %commentLine  '//'      ;
14 %commentBlock '/*' '*/' ;
15
16 Expr   -> Expr '+' Term
17         | Expr '-' Term
18         | Term ;
19
20 Term   -> Term '*' Factor
21         | Term '/' Factor
22         | Factor ;
23
24 Factor -> '-' Factor
25         | Int
26         | Name
27         | '(' Expr ')' ;
28
29 Actions
30
31 Arith +> Expr
32 Add x y  +>  (x +> Expr) '+' (y +> Term);
33 Sub x y  +>  (x +> Expr) '-' (y +> Term);
34 arith    +>  (arith +> Term);
35
36 Arith +> Term
37 Mul x y  +>  (x +> Term) '*' (y +> Factor);
38 Div x y  +>  (x +> Term) '/' (y +> Factor);
39 arith    +>  (arith +> Factor);
40
41 Arith +> Factor
42 Sub (Num 0) y  +>  '-' (y +> Factor);
43 Num i          +>  (i +> Int);
44 Var n          +>  (n +> Name);
45 arith          +>  '(' (arith +> Expr) ')';
```

Figure 2. A BIYACC program for the expression example

```
         Program ::= ‘Abstract’ HsDeclarations
                     ‘Concrete’ ProductionGroup+
                     ‘Actions’  ActionGroup+
 ProductionGroup ::= Nonterminal ‘->’ ProductionBody+{‘|’} ‘;’
  ProductionBody ::= Symbol+
          Symbol ::= Primitive | Terminal | Nonterminal
     ActionGroup ::= HsType ‘+>’ Nonterminal
                     Action+
          Action ::= HsPattern ‘+>’ Update+ ‘;’
          Update ::= Symbol
                   | ‘(’ HsVariable ‘+>’ UpdateCondition ‘)’
                   | ‘(’ Nonterminal ‘->’ Update+ ‘)’
 UpdateCondition ::= Symbol
                   | ‘(’ Nonterminal ‘->’ UpdateCondition+ ‘)’
```

Figure 3. Syntax of BIYACC programs (Nonterminals with prefix *Hs* denote Haskell entities and follow the Haskell syntax; the notation $nt^{+}\{sep\}$ denotes a nonempty sequence of the same nonterminal *nt* separated by *sep*.)

For reference, we give the definition of BIYACC syntax in Figure 3, which will also be referred to in Section 4.

2.1 Definitions of the abstract and concrete syntax

A BIYACC program consists of definitions of the abstract and concrete syntax and *actions* for reflectively printing ASTs to CSTs. The abstract syntax part — which starts with the keyword `Abstract` — is just one or more definitions of Haskell datatypes. In our example, the abstract syntax is defined in lines 1–9 by a single datatype `Arith` whose elements are constructed from constants, variables, and the arithmetic operators.

On the other hand, the concrete syntax — which is defined in the second part beginning with the keyword `Concrete` — is defined by a context-free grammar. For our expression example, in lines 16–27 we use a standard grammatical structure to encode operator precedence and order of association, which involves three nonterminal symbols `Expr`, `Term`, and `Factor`: An `Expr` can produce a left-leaning tree of `Term`s, each of which can in turn produce a left-leaning tree of `Factor`s. To produce right-leaning 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 is also a nonterminal *Name*, which produces identifiers.)

Syntax for comments. Syntax for comments can be declared at the beginning of this part before any production rules. For example, line 13 shows that the syntax for a single line comments is “`//`”, while line 14 states that “`/*`” and “`*/`” are respectively the beginning mark and ending mark for a block comment.

2.2 Actions

The last and 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 29–45 in Figure 2. Before explaining the actions, we should first emphasise that we are identifying program text with CSTs: 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 Section 4.2.1.

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

Let us look at a specific action — the first one for the expression example, at line 32 of Figure 2:

```
Add x y +> (x +> Expr) '+' (y +> Term);
```

The AST pattern is just a Haskell pattern; as for the CST 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 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 datatypes such as `Int`. 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 pattern is then created, and after that the whole group of actions is tried again. This time the pattern matching should succeed at the action used to create the CST, and the program will be able to make further progress.

Layout and comment preservation. The reflective printer generated by BiYacc is capable of preserving layouts and comments, but, perhaps mysteriously, in Figure 2 there is no clue as to how layouts and comments are preserved. This is because we decide to hide layout preservation from the programmer, so that the more important logic of abstract and concrete syntax synchronisation is not cluttered with layout preserving instructions. Our current 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.[1] 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. More details about this treatment and some exceptions can be found in Section 4.

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 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.

3. Foundation for BiYacc: putback-based bidirectional transformations

The behaviour of BiYacc is totally nontrivial: Not only do we need to generate two different programs from one, but 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 Section 1. It is possible to separately implement the *print* and *parse* semantics 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). (See Czarnecki et al. (2009) and Hu et al. (2011) for a comprehensive introduction.)

3.1 Parsing and printing as bidirectional transformations

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 of asymmetric lenses (Foster et al. 2007):

[1] One might argue, though, that layouts and comments should in fact be handled differently, since comments are usually attached to some entities which they describe. For example, when a function declaration is moved to somewhere else (e.g., by a refactoring tool), we will want the comment describing that function to be moved there as well. We leave the proper treatment of comments as future work.

A (well-behaved) *lens* between a *source* type S and a *view* type V is a pair of functions *get* and *put*, where

- the function $get :: S \to \mathsf{Maybe}\ V$ extracts a part of a source of interest to the user as a view, and
- the function $put :: S \to V \to \mathsf{Maybe}\ S$ takes a source and a view and produces an updated source incorporating information from the view.

Partiality is explicitly represented by wrapping the result types in the Maybe monad. The pair of functions should satisfy the *well-behavedness* laws:

$$put\ s\ v = \mathsf{Just}\ s' \quad\Rightarrow\quad get\ s' = \mathsf{Just}\ v \qquad (\textsc{PutGet})$$

$$get\ s = \mathsf{Just}\ v \quad\Rightarrow\quad put\ s\ v = \mathsf{Just}\ s \qquad (\textsc{GetPut})$$

Informally, 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 BX, satisfying the PUTGET and GETPUT laws by definition. The well-behavedness laws are then exactly the consistency properties (1) and (2) reformulated for a partial setting.

3.2 Putback-based bidirectional programming

Having rephrased parsing and printing in terms of BXs, we can now easily 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 well-behaved BX. Specifically, BIYACC programs are compiled to the *putback-based* bidirectional programming language BIGUL (Ko et al. 2016). It has been formally verified in AGDA (Norell 2007) that BIGUL programs always denote well-behaved BXs, and BIGUL has been ported to HASKELL as an embedded DSL library, which will be introduced in more detail in Section 3.3. 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 following theorem (Foster 2009):

Theorem. *Given a put function, there is at most one get function that forms a (well-behaved) BX with this put function.*

That is, once we describe a *put* function in BIGUL, not only can we immediately obtain a *get* function satisfying PUTGET and GETPUT with the *put* function, but also guarantee that the *get* function is unique, i.e., completely determined by the *put* function. Due to this theorem, in this paper we can focus solely on the printing (*put*) behaviour, leaving the parsing (*get*) behaviour only implicitly (but unambiguously) specified.

3.3 Bidirectional programming in BIGUL

Compilation of BIYACC to BIGUL (Section 4) only uses three BIGUL operations, which we explain here. A BIGUL program has type `BiGUL s v`, where `s` and `v` are respectively the source and view types; its *put* interpreter can then be given the type

```
put :: BiGUL s v -> s -> v -> Maybe s
```

Replace. The simplest BIGUL operation we use is

```
Replace :: 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,

```
put Replace _ v = v
```

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

```
$(update [p| spat |] [p| vpat |] [d| bs |]) :: 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, this `update` operation

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

pairs first component of the source with the view, since both are matched with `x`; the `Replace` operation is then performed on this pair, since it is the operation associated with `x` in the (singleton) list of operations[2]. In general, any (type-correct) BIGUL program can be used in the list of further updates, not just the primitive `Replace`. In the source pattern, the part marked by underscore (_) simply means that it will be skipped during the update.

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

```
Case :: [Branch s v] -> 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'*:

```
$(normalSV [p| spat |] [p| vpat |] [p| spat' |]) ::
BiGUL s v -> Branch s v
```

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

```
$(adaptiveSV [p| spat |] [p| vpat |]) ::
(s -> v -> s) -> Branch s v
```

[2] This representation of lists of named BIGUL operations is admittedly an abuse of syntax, but simplifies prototyping this system with Template Haskell.

Their *put* semantics 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 chosen branch is normal, the associated BIGUL operation is performed, and the updated source should satisfy the exit condition $spat'$ (otherwise it is a runtime error)[3]; when the chosen branch is adaptive, 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, and should go into a normal branch this time. Think of an adaptive branch as bringing a source that is too mismatched with the view to a suitable shape so that a normal branch — which 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.

4. Implementation of BIYACC

The architecture of BIYACC is illustrated in Figure 4. The programmer supplies a three-part BIYACC program as described in Section 2, which is compiled into an executable for converting between program text and ASTs. The conversion is essentially the composition of two bidirectional transformations, one between program text and CSTs and the other between CSTs and ASTs. The second BX is constructed as a BIGUL program, and is well-behaved by construction; we will explain in Section 4.1 how this BIGUL program is derived. The first BX, on the other hand, is in fact an isomorphism, which we construct in a more ad hoc manner. More specifically, we derive a lexer and a parser (using the parser generator HAPPY) for converting program text to CSTs, and a printer for flattening CSTs to program text; this isomorphism, which we call *concrete parsing and printing*, will be explained in Section 4.2. As is well known, isomorphisms are special cases of BXs, and the composition of two BXs is again a BX, so the composite transformations are still well-behaved.

Note that the grammar accepted by BIYACC is restricted to pure LALR (1) without disambiguation rules in the current implementation. And the well-behavedness of a BIYACC program is guaranteed by the compiled BIGUL program, which means that sometimes BIGUL will raise a runtime error for the bad transformations not satisfying the properties (1) and (2). In the future, we plan to extend BIYACC by making more static checks, and supporting disambiguation rules or employing other parser generators so that it can handle a wider class of grammars.

[3] The exit condition is an over-approximation of the range of this branch, so it is possible to check that the ranges of the branches in the same `Case` statement are disjoint, as is standard for bidirectional or reversible programs. In general, BIGUL's semantics incorporates several kinds of runtime checks for guaranteeing bidirectionality, and it is the programmer's responsibility to ensure that these checks can succeed.

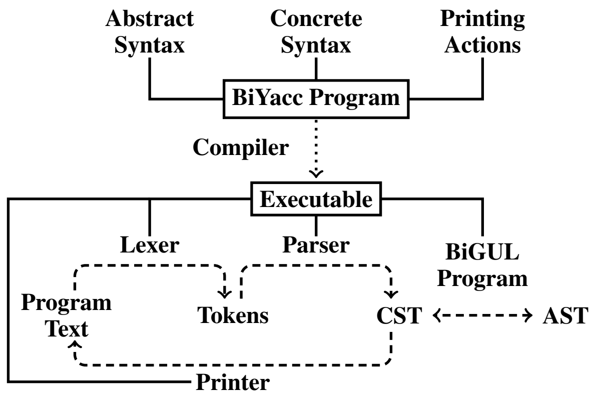

Figure 4. Architecture of BIYACC

4.1 Generating the BIGUL program

The semantics of BIYACC, shown in Figure 5, is defined by source-to-source compilation to BIGUL. Compilation rules are defined with the semantic bracket ($[\![\cdot]\!]$), 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, and additional arguments to the semantic bracket are typeset in superscript.

Top-level structure. A BIYACC *Program* has the form

`'Abstract'` *decls* `'Concrete'` *pgs* `'Action'` *ags*

and is compiled to a three-part Haskell program by copying *decls* (which is already valid Haskell code), converting each group of production rules in *pgs* to a datatype, and each action group in *ags* to a small BIGUL program. 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. Comment syntax declarations (`%commentLine` and `%commentBlock`) are only relevant to concrete parsing and printing, and are ignored in Figure 5.

CST datatypes. The production rules in a context-free grammar dictate how to generate strings from nonterminals, and a CST can be regarded as encoding one particular way of generating a string using the production rules. In Haskell, we represent CSTs starting from a nonterminal *nt* as a datatype named *nt*, whose constructors represent the production rules for *nt*. For each constructor we generate a unique name, which is denoted by $\textsc{con}(nt, syms)$. 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 are left unchanged (using their names for datatypes), terminal symbols are dropped, and an additional `String` field is added for terminals and primitives for storing layout information (whitespaces and comments) appearing after them in the program text. The last step is to insert an additional empty

$$\begin{aligned}
&\llbracket \texttt{‘Abstract’}\ \mathit{decls}\ \texttt{‘Concrete’}\ \mathit{pgs}\ \texttt{‘Action’}\ \mathit{ags} \rrbracket_{\mathit{Program}} = \\
&\quad \mathit{decls}\ \big\langle \llbracket \mathit{pg} \rrbracket_{\mathit{ProductionGroup}} \bigm| \mathit{pg} \in \mathit{pgs} \big\rangle\ \big\langle \llbracket \mathit{ag} \rrbracket_{\mathit{ActionGroup}} \bigm| \mathit{ag} \in \mathit{ags} \big\rangle
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathit{nt}\ \texttt{‘->’}\ \mathit{bodies} \rrbracket_{\mathit{ProductionGroup}} = \\
&\quad \texttt{‘data’}\ \mathit{nt}\ \texttt{‘=’} \\
&\qquad \big\langle \textsc{con}(\mathit{nt}, \mathit{syms})\ \big\langle \textsc{field}(s) \bigm| s \in \mathit{syms} \big\rangle\ \texttt{‘|’} \bigm| \mathit{syms} \in \mathit{bodies} \big\rangle \\
&\qquad \textsc{nullCon}(\mathit{nt})
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathit{vt}\ \texttt{‘+>’}\ \mathit{st}\ \mathit{acts} \rrbracket_{\mathit{ActionGroup}} = \\
&\quad \textsc{prog}(\mathit{vt}, \mathit{st})\ \texttt{‘::’}\ \texttt{‘BiGUL’}\ \mathit{st}\ \mathit{vt} \\
&\quad \textsc{prog}(\mathit{vt}, \mathit{st})\ \texttt{‘=’}\ \texttt{‘Case’} \\
&\quad\ \texttt{‘[’}\ \big\langle \llbracket a \rrbracket^{\mathsf{N}, \mathit{vt}, \mathit{st}}_{\mathit{Action}}\ \texttt{‘,’} \bigm| a \in \mathit{acts} \big\rangle\ \big\langle \llbracket a \rrbracket^{\mathsf{A}, \mathit{st}}_{\mathit{Action}} \bigm| a \in \mathit{acts} \big\rangle \{\texttt{‘,’}\}\ \texttt{‘]’}
\end{aligned}$$

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

$$\begin{aligned}
&\llbracket \texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{uc}_{\mathit{Primitive}}\ \texttt{‘)’} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} = \mathit{var}\ \texttt{‘= Replace;’} \\
&\llbracket \texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{uc}_{\mathit{Nonterminal}}\ \texttt{‘)’} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} = \\
&\quad \mathit{var}\ \texttt{‘=’}\ \textsc{prog}(\textsc{varType}(\mathit{vt}, \mathit{vpat}, \mathit{var}), \mathit{uc})\ \texttt{‘;’} \\
&\llbracket \texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \texttt{‘(’}\ \mathit{nt}\ \texttt{‘->’}\ \ldots \texttt{‘)’}\ \texttt{‘)’} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} = \\
&\quad \llbracket \texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{nt}\ \texttt{‘)’} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} \\
&\llbracket \texttt{‘(’}\ \ldots\ \texttt{‘->’}\ \mathit{updates}\ \texttt{‘)’} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} = \big\langle \llbracket u \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}}\ \texttt{‘;’} \bigm| u \in \mathit{updates} \big\rangle \\
&\llbracket \mathit{symbol} \rrbracket^{\mathit{vt}, \mathit{vpat}}_{\mathit{Update}} = \texttt{‘’}
\end{aligned}$$

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

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

$$\begin{aligned}
&\textsc{eraseVars}(\texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{uc}\ \texttt{‘)’})_{\mathit{Update}} = \mathit{uc} \\
&\textsc{eraseVars}(\texttt{‘(’}\ \mathit{nt}\ \texttt{‘->’}\ \mathit{updates}\ \texttt{‘)’})_{\mathit{Update}} = \\
&\quad \texttt{‘(’}\ \mathit{nt}\ \texttt{‘->’}\ \big\langle \textsc{eraseVars}(u) \bigm| u \in \mathit{updates} \big\rangle\ \texttt{‘)’} \\
&\textsc{eraseVars}(\mathit{symbol})_{\mathit{Update}} = \mathit{symbol}
\end{aligned}$$

$$\begin{aligned}
&\textsc{srcCond}(\texttt{‘(’}\ \mathit{nt}\ \texttt{‘->’}\ \mathit{uconds}\ \texttt{‘)’})_{\mathit{UpdateCondition}} = \\
&\quad \texttt{‘(’}\ \textsc{con}(\mathit{nt}, \big\langle \textsc{condHead}(\mathit{uc}) \bigm| \mathit{uc} \in \mathit{uconds} \big\rangle) \\
&\qquad \big\langle \textsc{srcCond}(\mathit{uc}) \bigm| \mathit{uc} \in \mathit{uconds} \big\rangle\ \texttt{‘)’} \\
&\textsc{srcCond}(\mathit{symbol})_{\mathit{UpdateCondition}} = \texttt{‘\_’}
\end{aligned}$$

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

$$\begin{aligned}
&\textsc{srcPat}(\texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{uc}_{\mathit{Primitive}}\ \texttt{‘)’})_{\mathit{Update}} = \texttt{‘(’}\ \mathit{var}\ \texttt{‘, \_)’} \\
&\textsc{srcPat}(\texttt{‘(’}\ \mathit{var}\ \texttt{‘+>’}\ \mathit{uc}_{\mathit{Nonterminal}}\ \texttt{‘)’})_{\mathit{Update}} = \mathit{var} \\
&\textsc{srcPat}(\texttt{‘(’}\ \mathit{nt}\ \texttt{‘->’}\ \mathit{updates}\ \texttt{‘)’})_{\mathit{Update}} = \\
&\quad \texttt{‘(’}\ \textsc{con}(\mathit{nt}, \big\langle \textsc{condHead}(\mathit{uc}) \bigm| \mathit{uc} \in \textsc{eraseVars}(\mathit{updates}) \big\rangle) \\
&\qquad \big\langle \textsc{srcPat}(u) \bigm| u \in \mathit{updates} \big\rangle\ \texttt{‘)’} \\
&\textsc{srcPat}(\mathit{symbol})_{\mathit{Symbol}} = \texttt{‘\_’}
\end{aligned}$$

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

Figure 5. Semantics of BIYACC programs (as BIGUL programs)

constructor, the unique name generated for which is denoted by $\textsc{nullCon}(nt)$; this empty constructor is used as a default value to a BIYACC printer whenever we want to create a new piece of program text depending on the view only.

For instance, the third group of the concrete syntax defined in Figure 2 is translated to the following Haskell declarations:

```
data Factor = Factor0 String Factor
            | Factor1 (Int, String)
            | Factor2 (Name, String)
            | Factor3 String Expr String
            | FactorNull2
```

Note that the first `String` field of `Factor0` stores the whitespaces appearing after a negation sign in the program text.

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 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. We said in Section 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 entry 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

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

is compiled to

```
$(normalSV [p| (Factor0 _ _) |] [p| Sub (Num 0) y |]
           [p| (Factor0 _ _) |])
  $(update [p| Sub (Num 0) y |] [p| (Factor0 _ y) |]
           [d| y = bigulArithFactor; |])
```

When the CST is a `Factor0` 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 $\textsc{varType}$) 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, the first action in the `Arith`–`Factor` group is compiled to

```
$(adaptiveSV [p| _ |] [p| Sub (Num 0) _ |])
  (\ _ _ ->  Factor0 " " FactorNull2)
```

The body of the adaptation function is generated by the auxiliary function defaultExpr, which creates a skeletal value that matches the source pattern.

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

```
entrance = bigulArithExpr
```

which is invoked in the `main` program.

4.2 Generating the concrete parser and printer

Having explained the bidirectional transformation between CSTs and ASTs, the remaining task is to establish an isomorphism between program text and CSTs.

4.2.1 The inverse properties

Assuming that the grammar is unambiguous and the parser generator is "correct", we will show that there exists an isomorphism between program text and CSTs:

$$\mathit{parse}\ \mathit{text} = \mathsf{Just}\ \mathit{cst} \quad \Rightarrow \quad \mathit{print}\ \mathit{cst} = \mathit{text} \tag{3}$$

$$\mathit{parse}\ (\mathit{print}\ \mathit{cst}) = \mathsf{Just}\ \mathit{cst} \tag{4}$$

One direction is a partial (Maybe-valued) function *parse* that converts program text into CSTs according to the grammar, and the other is a total function *print* from CSTs to program text.

To see what (3) means, we should first clarify what *print* does: Our CSTs, as described in Section 4.1, encode precisely the derivation trees, with the CST constructors representing the production rules used; what *print* does is simply traversing the CSTs and applying the encoded production rules to produce the derived program text. Now consider what *parse* is supposed to do: It should take a piece of program text and find a derivation tree for it, i.e., a CST which *print*s to that piece of program text. This statement is exactly (3). In other words, (3) is the functional specification of parsing, which is satisfied if the parser generator we use behaves correctly.

For (4), since the grammar is unambiguous, for any piece of program text there is at most one CST that prints to it, which is equivalent to saying that *print* is injective. In addition, it is reasonable to expect that a generated parser will be able to successfully parse any valid program text; that is, for any *cst* we have

$$\mathit{parse}\ (\mathit{print}\ \mathit{cst}) = \mathsf{Just}\ \mathit{cst}'$$

for some *cst′*. This is already close to (4); It remains to show that *cst′* is exactly *cst*, which is indeed the case because

$$\begin{aligned} & \mathit{parse}\ (\mathit{print}\ \mathit{cst}) = \mathsf{Just}\ \mathit{cst}' \\ \Rightarrow\ & \{ (3) \} \\ & \mathit{print}\ \mathit{cst}' = \mathit{print}\ \mathit{cst} \\ \Rightarrow\ & \{ \mathit{print} \text{ is injective} \} \\ & \mathit{cst}' = \mathit{cst} \end{aligned}$$

4.2.2 Concrete lexer and parser

In current BiYacc, the implementation of the *parse* function is further separated into two phases: tokenising and parsing. In both phases, the layout information (whitespaces and comments) is automatically preserved, which makes the CSTs isomorphic to the program text.

Lexer. Apart from handling the terminal symbols appearing in a grammar, the lexer automatically derived by BiYacc can also recognise several kinds of literals, including integers, strings, and identifiers, respectively produced by the nonterminals `Int`, `String`, and `Name`. For now, the forms of these literals are pre-defined, 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 considered as separators between other tokens. (Although there exist some languages such as Haskell and Python where indentation does affect the meaning of a program, there are workarounds, e.g., writing a preprocessing program to insert explicit separators.) Usually, token separators are thrown away in the lexing phase, but since we want to keep layout information in CSTs, which are built by the parser, the lexer should leave the separators intact and pass them to the parser. The specific approach taken by BiYacc is wrapping a lexeme and the whitespaces following it into a single token. Beginning whitespaces are treated separately from lexing and parsing, and are always preserved. And in this prototype implementation, comments are also considered as whitespaces.

Parser. The concrete parser is used to generate a CST from a list of tokens according to the production rules in the grammar. Our parser is built using the parser generator Happy, which takes a BNF specification of a grammar 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             : token0 Factor
                                { Factor0 $1 $2 }
   | Int                    | tokenInt
                      ⇝         { Factor1 $1 }
   | Name                   | tokenName
                                { Factor2 $1 }
   | '(' Expr ')';          | token1 Expr token2
                                { Factor3 $1 $2 $3 }
```

We use the first expansion (`token0 Factor`) to explain how whitespaces are transferred: The generated HAPPY token `token0` 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 `Factor0` by the associated HASKELL action.

5. Case study

The design of BIYACC may look simplistic and make the reader wonder how much it can describe. However, in this section we demonstrate with a larger case study that, without any extension, BIYACC can already handle real-world language features. For this case study, we choose the TIGER language, which is a statically typed imperative language first introduced in Appel's textbook on compiler construction (Appel 1998). Since TIGER's purpose of design is pedagogical, it is not too complex and yet covers many important language features including conditionals, loops, variable declarations and assignments, and function definitions and calls. TIGER is therefore a good case study with which we can test the potential of our BX-based approach to constructing parsers and 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 et al. (1988) and the draft version of 1999 ISO C standard, excluding the preprocessing part). The difference is basically due to the fact that C has more primitive types and various kinds of assignment statements.

Excerpts of the abstract and concrete syntax of TIGER are shown in Figure 6. The abstract syntax is substantially 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 (2013)'s lecture notes. Some changes are made to handle features that are not supported by current BIYACC: For example, to make the grammar become unambiguous without disambiguation rules, the operators are divided into several groups, with the highest-precedence terms (like literals) placed in the last group, just like what we did in the arithmetic expression example (Figure 2); the AST constructors `TFunctionDec` or `TTypeDec` take a single function or type declaration instead of a list of adjacent declarations (for representing mutual recursion) as in Appel (1998), since we cannot handle the synchronisation between a list of lists (in ASTs) and a list (in CSTs) with BIYACC's current syntax; finally, to circumvent the "dangling else" problem, a terminal "`end`" is added to mark the end of an `if-then` expression.

```
Abstract

data TExp = TString String | TInt Int | TNilExp
          | TSeq [TExp]    | TLet [TDec] TExp
          | TIf TExp TExp (Maybe TExp)
          | TOp TExp TOper TExp | ...
  deriving (Show, Eq, Read)
data TOper = TPlusOp | TMinusOp | ...
           | TEqOp | TNeqOp | ...
  deriving (Eq,Show,Read)

data TDec = TTypeDec TTyDec | TFunctionDec TFundec
  |TVarDec TSymbol (Maybe TSymbol) TExp
  deriving (Eq,Show,Read)
...
type TSymbol = String

Concrete

Exp -> 'break'  | LetExp | ArrExp | Assignment
     |  ForExp  | RecExp | IfThen | IfThenElse
     | WhileExp | PrimitiveOpt ;

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

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

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

PrimitiveOpt  -> PrimitiveOpt '|' PrimitiveOpt1
               | PrimitiveOpt1 ;
...
PrimitiveOpt3 -> PrimitiveOpt3 '+' PrimitiveOpt4
               | ... | PrimitiveOpt4 ;
...
PrimitiveOpt5 -> 'nil'  | Int    | String
               | LValue | SeqExp | CallExp
               | '-' PrimitiveOpt5 ;

IfThen     -> 'if' Exp 'then' Exp 'end' ;
IfThenElse -> 'if' Exp 'then' Exp 'else' Exp ;
...
```

Figure 6. An excerpt of TIGER's abstract and concrete syntax

Even though the underlying BIGUL programs have runtime checks as mentioned in the beginning of Section 4, we have successfully tested our BIYACC program for TIGER on all the sample programs provided on the homepage of Appel's book[4], including a merge sort implementation and an eight-queen solver, and there is no problem parsing and printing them. In the following subsections, we will highlight some nontrivial printing strategies used in that program to demonstrate what BIYACC, in particular reflective printing, can achieve.

5.1 Syntactic sugar

Syntactic sugar, which lets the programmer use some features in an alternative (perhaps conceptually higher-level) syntax,

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

is pervasive in programming languages. 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 `if` expressions in the abstract syntax: `e1 & e2` is desugared and parsed to `TIf e1 e2 (TInt 0)` and `e1 | e2` to `TIf e1 (TInt 1) e2`. The printing actions for them in BIYACC are:

```
TExp +> PrimitiveOpt
TIf e1 (TInt 1) (Just e2)     +>
 (e1 +> PrimitiveOpt) '|' (e2 +> PrimitiveOpt1);

TExp +> PrimitiveOpt1
TIf e1 e2 (Just (TInt 0))  +>
 (e1 +> PrimitiveOpt1) '&' (e2 +> PrimitiveOpt2);
```

The *parse* function for these syntactic sugar is not injective, since the alternative syntax and the features being desugared into are both mapped to the latter. A conventional printer — which takes only the AST as input — cannot reliably determine whether an abstract expression should be ensugared or not, whereas a reflective printer can make the decision by inspecting the CST.

5.2 Resugaring

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

We borrow Pombrio and Krishnamurthi's 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 5.1. For the "not" operator, which TIGER lacks, we introduce '~', represented by `TNot` in the abstract syntax. Now consider the source expression

```
~1 | ~0
```

which is parsed to

```
TIf (TNot (TInt 1)) (TInt 1) (J (TNot (TInt 0)))
```

where `J` is a shorthand for `Just`. A typical call-by-value evaluator will produce the following evaluation sequence given the above AST:

```
   TIf (TNot (TInt 1)) (TInt 1) (J (TNot (TInt 0)))
→ TIf (TInt 0) (TInt 1) (J (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 first 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's semantics is defined in terms of BIGUL, whose programs are always well-behaved. Also different from Pombrio and Krishnamurthi's approach is that we do not need to insert any additional information into the ASTs for remembering the form of the original sources. The advantage of our approach is that we can keep the abstract syntax pure, so that other tools — the evaluator in particular — can process the abstract syntax without being modified, whereas in Pombrio and Krishnamurthi's approach, the evaluator has to be adapted to work on the enriched abstract syntax.

Also note that the above resugaring for TIGER is achieved for free — the programmer does not need to write additional, special actions to achieve that. In general, BIYACC can easily and reliably reflect AST changes that involve only "simplification", i.e., replacing part of an AST with a simpler tree, so it should not be surprising that BIYACC can also reflect simplification-like optimisations such as constant propagation and dead code elimination, and some refactoring transformations such as variable renaming. All these can be achieved by one "general-purpose" BIYACC program, which does not need to be tailored for each application.

5.3 Language evolution

We conclude this section by looking at a practical scenario in language evolution, incorporating all the applications we introduced in this section. When a language evolves, some new features of the language (e.g., `foreach` loops in Java 5) can be implemented by desugaring to some existing features (e.g., ordinary `for` loops), so that the compiler does not need to be extended to handle the new features. As a consequence, all the engineering work about refactoring or optimising transformations 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 (1975)'s guarded command language. We extend TIGER's concrete syntax with the following production rules:

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

For simplicity, we restrict the predicate produced by `CaseB` to the form `LValue '=' Int`, but in general this 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
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing
  +>  'guard' (CaseBs -> (CaseB ->
         (lv +> LValue) '=' (i +> Int) '->' (e1 +> Exp))
            ) 'end';
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TIf _ _ _))
  +>  'guard' (CaseBs -> (CaseB ->
         (lv +> LValue) '=' (i +> Int) '->' (e1 +> Exp))
                          (if2 +> CaseBs)
            ) 'end';
```

```
TExp +> CaseBs
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing
    +>  (CaseB ->
          (lv +> LValue) '=' (i +> Int) '->' (e1 +> Exp));
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TIf _ _ _))
    +>  (CaseB ->
          (lv +> LValue) '=' (i +> Int) '->' (e1 +> Exp))
          (if2 +> CaseBs);
```

Though 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

```
TIf (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (J
  (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J
    (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16)
      Nothing))))
```

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 the AST will then be:

```
   TIf (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (J
    (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J
     (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16)
       Nothing))))
→ TIf (TInt 0) (TInt 4) (J
    (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J
     (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16)
       Nothing))))
→ TIf (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J
    (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16)
      Nothing))
→ TIf (TInt 1) (TInt 8) (J
   (TIf (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16)
     Nothing))
→ 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, 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 reflected to the surface. In our example, the first and third steps in the `TIf`-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 `TIf`.

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 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. We intend to handle this problem in the future.

6. Related work

Comparison with our earlier prototype. This paper is primarily based on and significantly extends our previous tool demonstration paper (Zhu et al. 2015), which conducts an early experiment with the idea of casting parsing and reflective printing in the framework of bidirectional transformations. Compared to our previous prototype, the current system has the following improvements:

- The well-behavedness of the underlying bidirectional language BiGUL (Section 3) we use in this version is formally verified (Ko et al. 2016), so we can guarantee the well-behavedness of BiYacc programs much more confidently;
- concrete parsers and printers between program text and CSTs are automatically derived (Section 4.2), so BiYacc synchronises program text and ASTs, not just CSTs and ASTs as with the previous version;
- we present many nontrivial applications such as resugaring and language evolution (Section 5);
- our current system tries to preserve layouts and comments in a transparent manner.

Unifying parsing and printing. Much research has been devoted to describing parsers and printers in a single program. For example, both Rendel and Ostermann (2010) and Matsuda and Wang (2013) adopt a combinator-based approach, where small components describing both parsing and printing are glued together to yield more sophisticated behaviour, and can guarantee inverse properties similar to (3) and (4) by construction (with CST replaced by AST in the equations). By specialising one of the syntax to be XML syntax, Brabrand et al. (2008) present a tool XSugar that handles bijection between the XML syntax and any other syntax for a grammar, guaranteeing that the transformation is reversible. However, the essential factor that distinguishes our system from others is that BiYacc is designed to handle synchronisation while others are all targeted at dealing with transformations.

Just like BiYacc, all of the systems described above handle pure unambiguous grammars without disambiguation declarations only. When the user-defined grammar (or the derived grammar) is ambiguous, the behaviour of these systems is as follows: Neither of Rendel and Ostermann's system (called "invertible syntax descriptions", which we shorten to ISDs henceforth) and Matsuda and Wang's system (called FliPpr) will notify the user that the (derived) grammar is ambiguous. For ISDs, property (4) will fail, while for FliPpr both properties (3) and (4) will fail. (Since the discussion on ambiguous grammars has not been presented in their papers, we try the examples provided by their libraries and find these problems.) In contrast, Brabrand et al. (2008) give a detailed discussion about ambiguity detection, and XSugar can statically check that 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 analysis is performed by the parser generator employed in the system, and the result is reported at compile time. If no warning is reported, the well-behavedness is always guaranteed, as explained in Section 4.2.1.

Even though all these systems handle unambiguous grammars only, there are design differences between them. 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. In BiYacc, the user defines the concrete syntax that has a hierarchical structure (`Expr`, `Term`, and `Factor`) to express operator priority, and write printing strategies to produce (preserve) necessary parentheses. The user of XSugar will also likely need to use such a hierarchical structure.

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

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

FliPpr will automatically expand the definition and derive a group of `ppr_i` functions indexed by the priority integer `i`, corresponding to the hierarchical grammar structure. In other words, there is no need to specify the concrete grammar, which is already implicitly embedded in the printer program. This makes FliPpr programs neat and concise. Following this approach, 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.

Bidirectional transformations (BXs). Our work is theoretically based on bidirectional transformations (Foster et al. 2007; Czarnecki et al. 2009; Hu et al. 2011), particularly taking inspiration from the recent progress on putback-based bidirectional programming (Pacheco et al. 2014a,b; Hu et al. 2014; Fischer et al. 2015; Ko et al. 2016). Also inspired by BXs, Martins et al. (2014) introduces an attribute grammar–based 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 *get* transformation, with a *put* transformation being automatically generated. One difference between Martins et al.'s system and ours is that their system performs synchronisation between two trees rather than program text and AST, and thus the preservation of layouts and comments is not considered. Another difference lies in the fact that, with their 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 such as BiYacc can give the user fuller control.

7. Conclusion

We have presented the design and implementation of BiYacc, with which the programmer can describe both a parser and a reflective printer for an unambiguous context-free grammar in a single program. Our solution is enriched by it's background of a putback-based bidirectional transformation framework which allows the guarantee of consistency properties by construction. This system can support various tasks of language engineering, from traditional constructions of basic machinery such as printers and parsers to more complex tasks such as Pombrio and Krishnamurthi (2014)'s "resugaring", language evolution and refactorings.

We plan to extend BiYacc to support disambiguation declarations. In many scenarios, it is more convenient for users to use an initially ambiguous grammar with additional disambiguation declarations such as precedences and order of association commonly found in parser generators such as Yacc and Happy. We are also constructing techniques to provide better usability to the tool, namely through the detection of possible inconsistencies on the user provided inputs which can now only be detected at runtime.

Acknowledgments

This work was partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009.

References

A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

R. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report Number 390, Computer Laboratory, University of Cambridge, 1966.

C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4):385–406, 2008.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Model Transformation*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.

M. De Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance*, pages 550–559. IEEE, 2002.

M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In *International Conference on Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2012.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8): 453–457, 1975.

J. Duregård and P. Jansson. Embedded parser generators. In *Haskell Symposium*, pages 107–117. ACM, 2011.

S. Fischer, Z. Hu, and H. Pacheco. The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21, 2015.

J. N. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

M. Hirzel and K. H. Rose. Tiger language specification. https://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001/tiger-spec.pdf, 2013.

Z. Hu, A. Schurr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar on bidirectional transformations (BX). *ACM SIGMOD Record*, 40(1):35–39, 2011.

Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In *International Symposium on Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2014.

B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

H.-S. Ko, T. Zan, and Z. Hu. BiGUL: A formally verified core language for putback-based bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 61–72. ACM, 2016.

J. Kort and R. Lämmel. Parse-tree annotations meet re-engineering concerns. In *International Workshop on Source Code Analysis and Manipulation*, pages 161–170. IEEE, 2003.

P. Martins, J. Saraiva, J. P. Fernandes, and E. Van Wyk. Generating attribute grammar-based bidirectional transformations from rewrite rules. In *Partial Evaluation and Program Manipulation*, pages 63–70. ACM, 2014.

K. Matsuda and M. Wang. FliPpr: A prettier invertible printing system. In *European Conference on Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2013.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for putback style bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014a.

H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *Principles and Practice of Declarative Programming*, pages 147–158. ACM, 2014b.

J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. *Programming Language Design and Implementation*, pages 361–371, 2014.

T. Rendel and K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Haskell Symposium*, pages 1–12. ACM, 2010.

T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002.

E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

Z. Zhu, H.-S. Ko, P. Martins, J. Saraiva, and Z. Hu. BiYacc: Roll your parser and reflective printer into one. In *International Workshop on Bidirectional Transformations*, pages 43–50. CEUR-WS, 2015.