A bidirectional language for parsing and reflective printing

Zirun Zhu  Yongzhe Zhang  Hsiang-Shang Ko
Pedro Martins  Joao Saraiva  Zhenjiang Hu

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, a domain-specific language BiYacc is proposed, with which users can specify both a parser and a reflective printer for an unambiguous grammar in a single program. 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. BiYacc is capable of facilitating many tasks such as Pombrio and Krishnamurthi's “resugaring”, language evolution, and refactoring.

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. A parser extracts the tree structure from such a string to a concrete syntax tree (CST), and converts it to an abstract syntax tree (AST), which is a

structured and simplified representation and is easier for 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, reporting internal information to the user, or observing the optimisations performed on the AST by the back-end of the compiler [25].

Parsers and printers do conversions in opposite directions and are intimately related — for instance, 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 both of the two components and keep them consistent with each other, which is a time-consuming and error-prone task. In response to this problem, many domain-specific
Original program text:
// the expression evaluates...
-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 an ideal printer:
// the expression evaluates...
-a /* a is the variable denoting. . . */ *
(2 + (a))

![Figure 1 Conventional printer and reflective printer](image)

Languages [3][28][26][9][21] have been proposed, in which the user can describe both a parser and a printer in a single program. By unifying these two pieces of software and deriving them from single and centralised code, we are creating a unified environment, which is easier to maintain and update, therefore respecting the “Don’t Repeat Yourself” principle of software development [16].

Despite their advantages, these domain-specific languages cannot deal with synchronisation between program text and ASTs, in the sense that a printer will always produce a new piece of program text from scratch. Let us look at a concrete example in Figure 1: The original program text is an arithmetic expression, containing negation, (redundant) parentheses, and some comments. 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 comments are 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 that is hard for humans to read and is usually 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 easier for the user to check where the changes are. With a conventional printer, immediately a problem occurs, as the printed result will likely mislead the programmer into thinking that the negation is replaced by a subtraction by the compiler; in addition, since the comments are 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 [7][19], and
- in program refactoring where the comments and layouts in the original program are completely lost after the AST is transformed by the system [6].

To make the printed result better, many attempts have been made to enrich ASTs with more information. To see a particular instance, Pombrio and Krishnamurthi [25] propose the notion resugaring, which devotes to representing evaluation sequences in a core language in terms of its surface syntax. Despite the fact that by applying these methods, the syntactic sugar (-a) in this example can be preserved during the printing, these methods contaminate ASTs by injecting more than necessary information. As a results, the compiler is hard to perform optimisation, and other tools designed for the language must also carefully handle the additional information in AST.

We aim to address these problems:
- unifying the design of parsers and printers,
- reflecting changes properly into the original
program text when printing, and

- keeping the AST clean and neat.

by proposing a domain-specific language BiYACC, which lets the user describe both a parser and a reflective printer for an unambiguous grammar in a single program. Different from a conventional printer, a reflective printer will take 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 can all be preserved. A taste of the reflective printing can be seen clearly from 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 a reflective printer is able to accept an AST and an empty piece of program text, in which case it will behave as 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.,

$$\text{parse (print s t)} = t \quad (1)$$

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.,

$$\text{print s (parse s)} = s \quad (2)$$

These two properties are inspired by the theory of bidirectional transformations [5][15], and are guaranteed by construction for all BiYACC programs.

An online tool that implements the approach described in this paper can be accessed at http://www.prg.nii.ac.jp/project/biyacc.html, which contains two input examples with various test cases used in this paper.

The rest of the paper is organised 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 [18], 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 [1], which shares many similarities with full grown, widely used languages. We demonstrate that BiYACC can handle syntactic sugar, partially subsume Pombrico and Krishnamurthi’s “resugaring” [25], and facilitate language evolution.
- Related work including detailed comparison with other systems is presented in Section 6 and in Section 7 we conclude the paper and present some future work.

2 Language design of BiYacc

This section presents the language design of BiYACC by a running example handling arithmetic expressions previously shown in Figure 1. We briefly summarise the definition of the syntax of BiYACC programs in Section 2.1. In Section 2.2, we present the structure, the syntax, and the semantics of BiYACC programs from a high-level aspect by writing the program for the arithmetic expression problem.
2.1 Definition of BiYacc’s syntax

The syntax of BiYacc programs is shown in Figure 2, where the definition is rather straightforward and can be treated as production rules. Anything wrapped in single quotes is string literal, which is basically keywords and built-in symbols of BiYacc. Terminals are usually keywords and symbols in a user-defined language (such as ‘+’, ‘-’, ‘let’, etc) and should be put within single quotes in a BiYacc program. A Nonterminal is a symbol that could be expanded to a list of Symbols in the usual sense. HsDeclarations are Haskell declarations, and HsPattern and HsType denotes respectively a Haskell pattern and a Haskell datatype that should conform to the datatype definitions in the HsDeclarations part. Their definitions follow the syntax of Haskell and are thus omitted here. A rare notation can be found in ProductionBody+{‘|’} where the bracket part means that one or more occurrence of ProductionBody is divided by the vertical bar (‘|’). Spaces (including newline) merely means string concatenation.

2.2 Programs in BiYacc

A BiYacc program consists of three parts: abstract syntax definition, concrete syntax definition, and actions describing how to update a concrete syntax tree (CST) with an AST.

2.2.1 Defining the abstract and concrete syntax

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 integer constants, variables, and the arithmetic operators. Different constructors — namely Add, Sub, Mul, Div, and Num — are used to construct different kinds of expressions, and in the definition each constructor is followed by the types of arguments it takes. Hence the constructors Add, Sub, Mul, and Div take two subexpressions (of type Arith) as arguments, while the constructors Var and Num respectively takes an String and Int as their arguments. For instance, the expression “1 + 2 * 3 – 4” can be represented as this AST of type Arith:

\[
\text{Sub} \langle \text{Add} \langle \text{Num} \, 1 \rangle, \\
\langle \text{Mul} \langle \text{Num} \, 2 \rangle, \\
\langle \text{Num} \, 3 \rangle \rangle \\
\langle \text{Num} \, 4 \rangle
\]

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, i.e., a set of production rules specifying how nonterminal symbols can be expanded to sequences of terminal and nonterminal symbols. 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 Terms, each of which can in turn produce a left-
leaning tree of Factors. 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 be-

inning 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 begin-
ning mark and ending mark for a block comment.

```
Abstract
1
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     | deriving (Show, Eq, Read)
9
Concrete
10
11 %commentLine '//' ;
12 %commentBlock '/*' '*/';
13 Expr -> Expr '+' Term
14     | Expr '-' Term
15     | Term ;
16 Term -> Term '*' Factor
17     | Term '/' Factor
18     | Factor ;
19 Factor -> '-' Factor
20     | Int
21     | Name
22     | '(' Expr ')';

3 Syntax definition parts of a BiYacc
program

2.2.2 Defining actions
The last and main part of a BiYACC program
starts with the keyword Actions, and describes how
to update a CST — i.e., a piece of program text —
with an AST. For our expression example, the
actions are defined in lines 29–45 in Figure 4. Be-
fore explaining the actions, we should emphasise
that we are identifying program text with CSTs:
Conceptually, whenever we write a piece of pro-
gram text, we are actually describing a CST rather
than just a sequence of characters. Technically, it
is almost effortless to convert a piece of program
text to a CST with existing parser technologies (we
follow the normal convention for parser-generating
systems that grammars must be unambiguous); the
reverse direction is even easier, requiring only a
traversal of the CST. By integrating with existing
parser technologies, BiYACC actions can focus on
describing conversions between CSTs and ASTs —
the more interesting part in the tasks of parsing and
printing. We will expound on this identification of
program text with CSTs in Section 4.3.

```
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);

34 Synchronisation strategy part of a BiYacc
program

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 4:

\[
\text{Add } x \ y \ +> (x \ +> \text{Expr}) \ +> (y \ +> \text{Term});
\]

For the view pattern Add x y, an AST (of type \(\text{Arith}\)) is said to match the pattern when it starts with the constructor Add (because the variable pattern, for example \(x\) and \(y\), matches anything); if the match succeeds, the two arguments of the constructor (i.e., the two subexpressions of the addition expression) are then respectively bound to the variables \(x\) and \(y\). As for the CST pattern, the main intention is to refer to the production rule \(\text{Expr} \rightarrow \text{Expr} \ +> \text{Term}\) and use it to match those CSTs produced by this rule. Since the action belongs to the group \(\text{Arith} \ +> \text{Expr}\), the part ‘\text{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 \(\text{Expr}\) and \(\text{Term}\) 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.

**Semantics of the program.**

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.

**Complex patterns.**

It is interesting to note that, by using more complex patterns, we can write actions that establish nontrivial relationships between CSTs and ASTs. For example, the action at line 42 of Figure 4 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 pattern: Suppose that we want to write a pattern that matches those CSTs produced by the rule \(\text{Factor} \rightarrow \text{-} \text{Factor}\), where the inner nonterminal \(\text{Factor}\) produces a further \(\text{-} \text{Factor}\) using the same rule. This pattern is written by overlaying the production rule on the first nonterminal \(\text{Factor}\) (an additional pair of parentheses is required for the expanded nonterminal): ‘\(\text{-}\) (\(\text{Factor} \rightarrow \text{-} \text{Factor}\)).

More examples involving this kind of deep patterns will be presented in Section 5.

**Layouts and comments preservation.**

The reflective printer generated by BiYacc is capable of preserving layouts and comments, but, perhaps mysteriously, in Figure 3 and Figure 4 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 [5] and [15] 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 [11]: A (well-behaved) lens between a source type \(S\) and a view type \(V\) is a pair of functions \(\text{get}\) and \(\text{put}\), where

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

\(^1\) 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.
Partiality is explicitly represented by wrapping the result types in the Maybe monad. The pair of functions should satisfy the well-behavedness laws:

\[ \text{put } s \, v = \text{Just } s' \Rightarrow \text{get } s' = \text{Just } v \ (\text{PutGet}) \]
\[ \text{get } s = \text{Just } v \Rightarrow \text{put } s \, v = \text{Just } s \ (\text{GetPut}) \]

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 [18]. It has been formally verified in Agda [22] that BiGUL programs always denote well-behaved BXs. 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 specific BiGUL that we used for the BiYacc implementation, is an embedded DSL library of the Haskell port version. Here we introduce three constructs in BiGUL to which a BiYacc program is compiled to.

A BiGUL program has type \( \text{BiGUL } s \, v \), where \( s \) and \( v \) are respectively the source and view types; its put interpreter can then be given the type

\[ \text{put} :: \text{BiGUL } s \, v \rightarrow s \rightarrow v \rightarrow \text{Maybe } s \]

**Replace.**

The simplest BiGUL operation we use is

\[ \text{Replace} :: \text{BiGUL } s \, s \]

which discards the original source and returns the view — which has the same type as the source — as the updated source. That is,

\[ \text{put } \text{Replace } s \, v = v \]

**Update.**

The next operation update is more complex, and is implemented with the help of Template Haskell [27]. The general form of the operation is

\[ $(\text{update } [p\,\text{pat }|\,|\,p\,\text{pat }|\,|\,d\,\text{bs }|]) \]
This operation decomposes the source and view by pattern matching with the patterns \textit{spat} and \textit{vpat} respectively, pairs the source and view components as specified by the patterns (see below), and performs further BiGUL operations listed in \textit{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

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

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

\textbf{Case.}

The most complex operation we use is \textit{Case} for doing case analysis on the source and view:

\[
\text{Case :: } [\text{Branch } s \ v] \rightarrow \text{BiGUL } s \ v
\]

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

\[
\text{\$\{normalSV [p| \text{spat } |] [p| \text{vpat } |] [p| \text{spat’ } |] \}: : \text{BiGUL } s \ v \rightarrow \text{Branch } s \ v}
\]

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

\[
\text{\$\{adaptiveSV [p| \text{spat } |] [p| \text{vpat } |] \}: :}
\]

Their \textit{put} semantics are as follows: A branch is applicable when the source and view respectively match \textit{spat} and \textit{vpat} in its main condition. Execution of a \textit{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 \textit{spat’} (otherwise it is a runtime error); when the chosen branch is adaptive, the associated function is applied to the source and view to compute an adapted source, and the whole \textit{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.

\section{Implementation of BiYacc}

This section provides a prototype implementation of BiYacc in detail. The architecture of BiYacc is shown in Figure 5: On the surface, the programmer supplies a three-part BiYacc program as described in Section 2.2, which is compiled into an executable for performing synchronisation between program text and ASTs. Since it is difficult to synchronise non-structured program text and structured ASTs directly, we make use of existing technologies and separate the synchronisa-

\footnote{This representation of lists of named BiGUL operations is admittedly an abuse of syntax, but simplifies prototyping this system with Template Haskell.}

\footnote{The \textit{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 \textit{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.}
tion into two phases. The synchronisation now becomes 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 3.2 the basics of BiGUL and 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 and Section 4.3. 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.

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 declarations or employing other parser generators so that it can handle a wider class of grammars.

### 4.1 Generating the BiGUL program

The semantics of BiYacc, shown in Figure 6, is defined by source-to-source compilation to BiGUL. Compilation rules are defined with the semantic bracket ([ [ ] ]), 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 \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 6.

**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 \( \text{cos}(nt, syms) \). 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 \( \text{field} \): Nonter-
'Abstract decls 'Concrete' decls 'Action' decls 'Program' =
  decls (pp) ProductionGroup | pp in pgs) (||ag) ActionGroup | ag in ags)

[nt ->' bodies] ProductionGroup =
  'data' nt :=' (con(nt, syms) (field(s) s in syms) '1' | syms in bodies) NullCon(nt)

[vt +-> st acts] ActionGroup =
  prog(st, st) : 'BIGUL' st st
  prog(st, st) = 'Case' (nt) st acts

[vpats updates] ActionGroup =
  'defaultExpr (pp [vpat] st acts)
  'eraseVars (pp [vpat] st acts)

FIELD(nt) Nonterminal = nt
FIELD(t) Terminal = 'String'
FIELD(p) Primitive = 'p', String'

erasevars('ct', uc) update = uc
erasevars('ct', nt updates') update =
  ('ct nt updates') erasevars(uc) uc in updates')
erasevars(symbol) update = symbol

srccond('ct', uc updates') updatecondition =
  (con(nt, condhead(uc)) uc in uc) srccond(uc uc in uc)
condhead(symbol) updatecondition = symbol

srcpat('ct', uc primitive) update = ('ct', uc)
srcpat('ct', uc nonterminal) update = var
srcpat('ct', uc updates') update =
  (con(nt, condhead(uc)) uc in erasevars(updates')) srcpat(uc uc in updates')

defaultexpr(symbol) primitive = ('undefined, '')'
defaultexpr(symbol) nonterminal = nullcon(symbol)
defaultexpr(symbol) terminal = 'ordo'
defaultexpr('ct', uc updates') updatecondition =
  (con(nt, condhead(uc) uc in uc updates')) defaultexpr(uc uc in uc)

6 Semantics of BiYacc programs (as BiGUL programs)

minals 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 ad-
ditional empty constructor, the unique name generated for which is denoted by \texttt{nullCon} (\texttt{Int}); 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 3 is translated to the following Haskell declarations:

\begin{verbatim}
data Factor = Factor0 String Factor
  | Factor1 (Int, String)
  | Factor2 (Name, String)
  | Factor3 String Expr String
  | FactorNull2
deriving (Show, Eq, Read)
\end{verbatim}

Note that the first \texttt{String} field of \texttt{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 \texttt{vt} and source type \texttt{st} and denoted by \texttt{prog(vt, st)}. The BiGUL program has one single \texttt{Case} statement, and each action is translated into two branches in this \texttt{Case} statement, one normal and the other adaptive. All the adaptive branches are gathered in the second half of the \texttt{Case} statement, so that normal branches will be tried first. For example, the third group of type \texttt{Arith \to Factor} is compiled to

\begin{verbatim}
t (normalSV [p| (Factor0 _ _) |] [p| Sub (Num 0) y |]
   [p| (Factor0 _ _) |])
t (update [p| Sub (Num 0) y |] [p| (Factor0 _ y) |]
   [d| y = bigulArithFactor; |])
\end{verbatim}

When the CST is a \texttt{Factor0} and the AST matches \texttt{Sub (Num 0) y}, we enter this branch, decompose the source and view by pattern matching, and use the view’s right subtree \texttt{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 \texttt{y} (deduced by \texttt{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 \texttt{Arith}–\texttt{Factor} group is compiled to

\begin{verbatim}
bigulArithFactor :: BiGUL Factor Arith
bigulArithFactor = Case [...] (adaptiveSV [p| _ |] [p| Sub (Num 0) _ |])
_ _ -> Factor0 " " FactorNull2
\end{verbatim}

The body of the adaptation function is generated by the auxiliary function \texttt{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

\begin{verbatim}
entrance = bigulArithExpr
\end{verbatim}

which is invoked in the \texttt{main} program.

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

\[
\text{parse text} = \text{Just cst} \quad \Rightarrow \quad \text{print cst} = \text{text} \quad (3)
\]

\[
\text{parse (print cst)} = \text{Just cst} \quad (4)
\]

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 the meaning of (3), 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 `prints` 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

\[
\text{parse (print cst)} = \text{Just cst}'
\]

for some `cst'`. This is already close to (4); what remains to be shown is that `cst'` is exactly `cst`, which is indeed the case because

\[
\text{parse (print cst)} = \text{Just cst'}
\]

\[
\Rightarrow \quad \{ (3) \}
\]

\[
\text{print cst'} = \text{print cst}
\]

\[
\Rightarrow \quad \{ \text{print is injective} \}
\]

\[
cst' = \text{cst}
\]

### 4.3 Generating concrete parser and printer

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. In the following, we will show how the lexer and parser are constructed and later the printer.

#### Generating 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 ex-
plicit 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 so that the CST still contains these whitespaces and a trivial printer can be constructed to naturally transform the CST back to original program text. Beginning whitespaces are treated separately from lexing and parsing, and are always preserved. And in this prototype implementation, comments are also considered as whitespaces.

Generating 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 : token0 Factor
  | Int | tokenInt
  | Name | tokenName
  | '(' Expr ')' | token1 Expr token2
FactorNull2
```

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 Factor by the associated Haskell action.

Generating printer.

As mentioned in Section 4.2, a printer generated by BiYacc is supposed to traverse the CST and apply the encoded production rules to retrieve that piece of program text, which is rather simple and straightforward. For example, the CST for Factor is flattened in the following way by defining the instance of Show class in Haskell:

```
instance Show Factor where
  show (Factor0 s0 s1) = "~-" ++ s0 ++ show s1
  show (Factor1 s0) = fst s0 ++ and s0
  show (Factor2 s0) = fst s0 ++ and s0
  show (Factor3 s0 s1 s2) = s0 ++ show s1 ++ s2
  show (FactorNull2) = ""
```

5 Case studies

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. Section 5.1 gives a brief introduction to the Tiger language, where we also make comparison with the C programming language to show the complexity of Tiger. In the following sections, some representative cases are presented to demonstrate what BiYacc and reflective printing can achieve, including preservation of syntactic sugar, constant propagation, Pombrio and Krishnamurthi’s resugaring [25], and language evolution.

5.1 The Tiger language

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 [1]. 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. Some of these features can be seen in this TIGER program:

```plaintext
def 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 an LALR grammar with 81 production rules to specify TIGER’s syntax, while for C89 and C99 it takes 183 and 237 rules respectively (based on [17] 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. 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.

Excerpts of the abstract and concrete syntax of TIGER are respectively shown in Figure 7 and Figure 8, where the usage of constructors can be guessed from its name easily. 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’s lecture notes [13]. 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 3); 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’s book, 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. One remark is that the language shown in Figure 3 is a subset of TIGER and is reused for building this example except for some changes on names of constructors and production rules, which means that BiYACC programs can be developed in an incremental way.

We have tested our BiYACC program for TIGER on all the sample programs provided on the home-page of Appel’s book†, including a merge sort implementation and an eight-queen solver, and there is no problem parsing and printing them. The complete BiYACC program for TIGER can be found

†4 https://www.cs.princeton.edu/~appel/modern/testcases/
Concrete

Exp -> 'break' | LetExp | ArrExp | Assignment
  | ForExp | RecExp | IIfThen | PrimitiveOpt
  | IIfThenElse | WhileExp ;

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

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

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

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

IIfThen -> 'if' Exp 'then' Exp 'end'
IIfThenElse -> 'if' Exp 'then' Exp 'else' Exp ;

BiYACC are:

TExp +> PrimitiveOpt
  TIf e1 (TInt 1) (Just e2) +>
    (e1 +> PrimOpt) '||' (e2 +> PrimOpt1);
  t +> (t +> PrimOpt1);

TExp +> PrimitiveOpt1
  TIf e1 e2 (Just TInt 0) +>
    (e1 +> PrimOpt1) '&^' (e2 +> PrimOpt2);
    t +> (t +> PrimOpt2);

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.

5.3 Constant propagation

Constant propagation is an important compiler optimisation that substitutes the values of known constants in expressions at compile time rather than at runtime. We use the example below of solving the eight-queen problem to show that BiYACC enables the observation of constant propagation performed on AST from program text.

Suppose the code snippet is:

let var N := 8
  type intArray = array of int
  var row := intArray [ N ] of 0
...
function try(c:int) = (...)
in try(0)
end

where the variable N denotes the size of the problem. Since the length of the arrays only depends on the variable N, it can be evaluated and optimised at compile time. The right hand side of the variable declaration at line 3 is parsed to:

TArrayExp "intArray" (TVarExp (TSimpleVar "N")) (TInt 0)

After constant propagation, it is optimised to:

TArrayExp "intArray" (TInt 8) (TInt 0)
Now we reflect the change back to program text using BiYacc by simply performing printing. The result is:

```
let var N := 8
  type intArray = array of int
  var row := intArray [8] of 0
  ...
  function try(c:int) = (...)
in  try(0)
end
```

Since the change is precisely located in the right hand side of the variable declaration at line 3, other parts of the program including layouts remain intact.

## 5.4 Resugaring

The idea of resugaring [25] 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 [25].

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 5.2. 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) (Just (TNot (TInt 0)))
```

A typical call-by-value evaluator will produce the following evaluation sequence given the above AST:

```
→ TIf (TInt 0) (TInt 1) (Just (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 their 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 their 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 (we have mentioned) 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.

## 5.5 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’s guarded command language [8]. 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 (Just if2@ (TIf _ _ _)) => 'guard' (CaseBs => (CaseB => (lv => LValue) '=' (i => Int) '->' (e1 => Exp)) (if2 => CaseBs)) 'end';
```

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 (TEq (TName "choice") (TInt 1)) (TInt 4)
  (TIf (TEq (TName "choice") (TInt 2)) (TInt 8)
    (TIf (TEq (TName "choice") (TInt 3)) (TInt 16)
      TSeqNil))
```

Suppose that the value of the variable choice is 2. The evaluation sequence on the the AST will then be:

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

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

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

```
TInt 8
```

And the reflected evaluation sequence on the con-
crete expression will be:

\[
\text{guard choice = 1 -> 4} \\
\text{choice = 2 -> 8} \\
\text{choice = 3 -> 16 end}
\]

Reflective printing fails for the first and third steps, but this behaviour in fact conforms to Pombrio and Krishnamurthi’s “abstraction” property [25], 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 \text{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 \text{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. This problem is discussed in Section 7.2.

6 Related work

Comparison with our earlier prototype.

This paper is primarily based on and significantly extends our previous tool demonstration paper [29], 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:

- concrete parsers and printers between program text and CSTs are automatically derived (Section 4.3), so \text{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 [26] and Matsuda and Wang [21] 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. [4] present a tool \text{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 \text{BiYacc} is designed to perform synchronisation while others are all targeted at handling transformations.

Just like \text{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 \text{FliPpr}) will notify the user that the (derived) grammar is ambiguous. For ISDs, property (4) will fail, while for \text{FliPpr} both properties (3) and (4) will fail. (Since the discus-
tion 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. [4] 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.

Even though all these systems handle unambiguous grammar 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) =
prensIf (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, 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 [11][5][15], particularly taking inspiration from the recent progress on putback-based bidirectional programming [23][24][14][10][18]. As explained in Section 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.
Get-based vs putback-based.

Martins et al. [20] introduces an attribute grammar–based BX system for defining transformations between two representations of languages (two grammars). The utilisation is similar to Bi-Yacc: 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 32–34 of Figure 3 can be expressed in Martins et al.’s system as

\[
\text{get}^E_{EA}(\text{add}(l, '+', r)) \rightarrow \text{plus} \left( \text{get}^E_{EA}(l), \text{get}^T_{TA}(r) \right)
\]

\[
\text{get}^E_{EA}(\text{sub}(l, '-r', r)) \rightarrow \text{minus} \left( \text{get}^E_{EA}(l), \text{get}^T_{TA}(r) \right)
\]

\[
\text{get}^E_{EA}(\text{et}(t)) \rightarrow \text{get}^T_{TA}(t)
\]

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 rules can be applied, Martins et al.’s system chooses, by design, the one that creates the most specialised version. This might or might not be ideal for the user of the system. 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 5.2, coexisting with a more general rule that maps a concrete if expression to an abstract if expression. Then printing the AST \( T\text{if} \left( \text{Name } "a" \right) \left( \text{Name } "b" \right) \theta \) from scratch will and can only produce \( a \& b \) or \( a \text{ then } b \text{ else } \theta \) by suitably ordering the actions.

This difference is somewhat subtle, and one might argue that Martins et al.’s design simply went one step too far — if their system had been designed to respect the rule ordering as specified by the user, as opposed to always choosing the most specialised rule, the system would have given its user the same flexibility as BiYACC. Interestingly, whether to let user-specified rule/action ordering affect the system’s behaviour is, in this case, exactly the line between get-based and 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. We can view this from another angle: If the user is required to specify a forward transformation while customising the backward behaviour by carefully ordering the rules, then the purpose of a bidirectionalisation system — which is to reduce the problem of writing bidirectional transformations to unidirectional programming — is largely defeated. 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. Of course, it would be rather disappointing if all we can gain from adopting the putback-based approach was just the ability to customise the behaviour of printing from scratch. We will make another case for putback-based thinking by sketching one potential extension to BiYACC in Section 7.2.
7 Conclusion and future work

In this paper, 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 its background of a putback-based bidirectional transformation framework which allows the guarantee of partial version of the consistency properties (1) and (2) 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’s “resugaring”, language evolution and refactorings.

We end this paper by presenting some inspiring future work. In Section 7.1, we analyse the tough problem caused by incorporating disambiguation declarations such as precedences and order of association commonly found in parser generators such as Yacc and Happy. In Section 7.2 we show the problem of naively matching-by-position strategy, and propose a new global matching strategy that can be achieved by carefully design adaptation branches, which require a new surface syntax to support however.

7.1 Handling ambiguous grammars

As explained in Section 4.2, we assume the grammars given to BiYacc are unambiguous in order to guarantee the inverse properties for concrete parsing and printing. In many scenarios, however, it is more convenient for the user to use an initially ambiguous grammar with additional disambiguation rules such as declarations of precedence and order of association. For instance, the declarations starting from %left, %right, and %nonassoc can be commonly found in parser generators such as Yacc and Happy, meaning that the following terminals’ order of association is respectively left-associative, right-associative, or non-associative. And the order of these declarations specify the precedence of these operators, with later declarations having higher precedence. It is therefore desirable to extend BiYacc to incorporate such use.

However, to use ambiguous grammars, we have to rethink about how to guarantee the inverse properties since the analysis in Section 4.2 is no longer applicable. Let us revisit the arithmetic expression example in Figure 3, and define the concrete syntax with additional disambiguation rules:

%left '\*' '\-';
%left '\*' '/';
Expr -> Expr '\*' Expr | Expr '\-' Expr
| Expr '\*' Expr | Expr '/' Expr
| Name | Int | '\(' Expr '\)'
| '\-' Expr;

For the concrete parser, a natural choice is to use the one generated by Happy from this grammar; this concrete parser will always construct valid CSTs, i.e., those CSTs that respect the precedence and order-of-association declarations. But then, for the printing direction, we can no longer use the primitive printer that strictly follows the production rules, since the inverse property (4) will be invalidated: If we print an invalid CST to a piece of program text and then parse the text, the resulting CST is necessarily valid and cannot be the one we started with.

One way out might be to restrict the domain of the primitive printer to accept only valid CSTs (and revise (4) to take additional partiality into account), but this can make it hard to arrive at correctly behaving actions. For example, suppose that we have the program text x*2/5, and optimise the “x*2” part in the corresponding AST to “x+x”, as shown on the left side below:
If the actions are written naively, merely relating corresponding AST and CST structures, then the updated AST will be printed by the BiGUL part to the invalid CST on the right above, which is rejected by the partial concrete printer, resulting in an unexpected printing failure. We thus conclude that for now there are complicated issues to be tackled if ambiguous grammars are to be supported and leave the work to the future.

### 7.2 Global matching strategies

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, and this may not be desirable.

A typical scenario where the matching of a CST and an AST can be easily disrupted is the synchronisation of two expression sequences. Consider the following example in Tiger:

```tiger
( k := 2; /* dead code */
  inc(x); /* increment on x */
  k := x + 5;
  if k then ...;
  ...
)
```

Suppose that the AST is optimised by removing the dead expression `k := 2`. Then the CST and updated AST become:

What we want is removing the first concrete subtree, synchronising the second concrete subtree with the first abstract subtree, and so on, in order to keep as many original whitespaces as possible (e.g., the comment `/* increment on x */`). However, the BiYACC program synchronises the concrete and abstract subtrees in order, and the subtrees are all mismatched, causing the concrete subtrees to be printed from scratch and the original whitespaces to be discarded. This scenario motivates us to extend BiYACC with a mechanism in which the user can write strategies that match subtrees of the same structure in a more global manner.

This matching problem is one that can be solved more effectively with the putback-based approach, since matching strategies inherently belong to the putback direction. Indeed, the global matching strategies such as based on the longest common subsequence of two expression sequences, or based on tree edit distance [2] can be hand-coded in BiGUL, however, it is still a challenge to design proper surface syntax in BiYACC for the user to write their own global matching strategies.

**Acknowledgement** 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**


