Noname manuscript No.
(will be inserted by the editor)

FliPpr: A System for Deriving Parsers from Pretty-Printers

Kazutaka Matsuda · Meng Wang

the date of receipt and acceptance should be inserted later

Abstract When implementing a programming language, we often write a parser and a pretty-printer. However, manually writing both programs is not only tedious but also error-prone; it may happen that a pretty-printed result is not correctly parsed. In this paper, we propose FliPpr, which is a program transformation system that uses program inversion to produce a CFG parser from a pretty-printer. This novel approach has the advantages of fine-grained control over pretty-printing, and easy reuse of existing efficient pretty-printer and parser implementations.

1 Introduction

In this paper, we will discuss the implementation of a programming language, say the following one

$$\begin{array}{lll} \mathit{prog} & ::= & \mathit{rule}_1; \ldots; \mathit{rule}_n \\ \mathit{rule} & ::= & f\ p_1\ \ldots\ p_n = e \\ p & ::= & x \mid \mathsf{C}\ p_1\ \ldots\ p_n \\ e & ::= & x \mid \mathsf{C}\ e_1\ \ldots\ e_n \mid e_1 \oplus e_2 \mid f\ e_1\ \ldots\ e_n \end{array}$$

This work was mainly done when the first author was at University of Tokyo, and the second author was at Chalmers University of Technology. We thank Nils Anders Danielsson for his critical yet constructive comments on an earlier version of this work, without which the surface language probably would not exist. We also thank Janis Voigtländer and Akimasa Morihata for their insightful comments on deforestation. This work was partially supported by JSPS KAKENHI Grant Numbers 24700020, 15K15966, and 15H02681. Part of this research was done when the first author was visiting Chalmers Univeristy of Technology supported by Study Program at the Overseas Universities by Graduate School of Information Science and Technology, the University of Tokyo.

K. Matsuda
Tohoku University. 6-3-09, Aza Aoba, Aramaki, Aoba-ku, Sendai, Miyagi 980-8579, JAPAN.

M. Wang
University of Bristol. Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.

which is a standard first-order functional language with data constructors C, functions f and binary operators $\oplus$. Ignoring the semantics of the language for the time being, we start with writing a parser and a pretty-printer to deal with the syntax: the parser converts textual representations of programs into the AST, and the pretty-printer converts the AST to nicely laid-out programs. Though not often measured objectively, the prettiness of printing results is important: a pretty-printer is central to the communication between a compiler and the programmers, and the quality of it directly contributes to the productivity and satisfaction of the users of the language.

Despite being developed separately, the parser and the pretty-printer are always expected to be consistent with each other: very informally, parsing a pretty-printed program should succeed, and produce the same AST that is pretty-printed. It is common knowledge that consistency properties like this between a pair of tightly-coupled programs are hard to produce and maintain; and perhaps less widely known that they are difficult to be tested effectively too, due to the complexity of AST data [**?**].

In this paper, we are going to discuss the implementation of a language, which has a more elaborated version of the above-presented syntax. The language can be used to program pretty-printers, and at the same time through program inversion techniques, obtain a consistent parser. We, as usual, manually implemented a parser and a pretty-printer for the language, but with the hope that we, and many others who read this paper, will not need to do it again for their own language implementations.

Prior to this work, there has been a rich body of literature on exploring correctness-by-construction techniques to automatically generate one or both programs of the printer/parser pair, notably [**?**, **?**, **?**] . We have intentionally omitted the prefix "pretty-" from the mentioning of printers here because few of the existing work actually produce pretty-printers in the sense of Hughes [**?**] and Wadler [**?**].[1]

To be more precise about what we mean by *prettiness*, let us consider a subtraction language $e ::= 1 \mid e_1 - e_2$ that has a constant (1) and a left-associative binary operator $(-)$. We represent the syntax with the following AST datatype.

$$\textbf{data}\ E = \mathsf{One} \mid \mathsf{Sub}\ E\ E$$

As an example, let's consider the expression Sub (Sub One One) (Sub One One). A not so good printer that performs a simple traversal of the tree may produce `(1 - 1) - (1 - 1)`, while a higher-quality printer may be able to spot the redundancy of parentheses on the left and produce `1 - 1 - (1 - 1)`. To be considered as a *p*retty-printer in the sense of Hughes [**?**] and Wadler [**?**], a printer is additionally required to have refined behaviour during line-wrapping. For example the above expression shall be pretty-printed as

[1] The Syn system [**?**] is capable of handling non-contextual layouts, which can be seen as a limited form of prettiness.

```
1 - 1 - (1 - 1)   or   1 - 1        or   1 - 1
                         - (1 - 1)           - (1
                                               - 1)
```

depending on the screen width that is used to render the result. This fine-grained control from users over bracketing, spacing and indentation is clearly beyond any technique based on mechanical traversals of ASTs, which is likely to rigidly produce `1 - 1 - (1 - 1)` (with arbitrary line-wrapping) or even `(1 - 1) - (1 - 1)` as the only printing result.

Knowing that prettiness cannot be generated automatically, various libraries have been developed for programming pretty-printers, most notably Hughes's [**?**] and Wadler's [**?**]. In this paper based on Wadler's library [**?**], we propose a novel system FliPpr (pronounced as "flipper"). In FliPpr, a programmer provides a pretty-printer carefully tuned for prettiness (which is slightly annotated with some additional information for parsing), and then the system inverts it to obtain a consistent parser. For example, a pretty-printer exhibiting the above behaviour can be defined as below.

```
ppr One         = text "1"
ppr (Sub e1 e2) = group (ppr e1 <> nest 2 (line <> text "-" <> text "  " <> pprP e2))
-- The suffix P in pprP stands for parentheses.
pprP One        = text "1"
pprP (Sub e1 e2) =
    text "(" <> group (ppr e1 <> nest 2 (line <> text "-" <> text "  " <> pprP e2)) <> text ")"
```

The pretty-printing library functions are shown in *slant sans serif*. As a general rule, infix operators binds looser than prefix applications. Roughly speaking, *text* s converts a string s to a layout, d_1 <> d_2 is an infix binary operator that concatenates two layouts d_1 and d_2. The nullary function *line* starts a new line, but its behavior can be affected by surrounding *nest* and *group* applications: *nest* n d inserts n-spaces after each *line*s in d, and *group* d smartly chooses between the layout d and other layouts derivable from d by selectively interpreting *line*s as single spaces.[2]

We claim the following benefits of our approach:

- **Fine-Grained Control over Pretty-Printing**. Our language based on Wadler's library [**?**] offers the possibility of refined control over different aspects of pretty-printing: spacing can be tuned; redundant bracketing can be eliminated through the passing of fixity and precedence information; indentation can be designed by nesting lines; and wrapping of lines can be performed smartly.
- **Efficiency**. FliPpr is efficient in the sense that we can reuse existing efficient implementation of pretty-printers and parsers. For pretty-printing, we can use Wadler's library [**?**] and other refined pretty-printing algorithms [**?**, **?**].

[2] In this paper, we write "space" for the space character and write "whitespace" for the space character and the new-line character. Other kinds of spaces such as horizontal tabs are not discussed as they do not yield new insight.

For parsing, we can use any parser generator that supports CFG without restriction.

The technique of program inversion used in FliPpr is not new; it is a direct consequence of our previous work [**?**]. The novelty of this paper lies in the design of the pretty-printing system, which makes the program inversion possible. Specifically, in this work:

- We propose an invertible pretty-printing technique based on grammar-based inversion [**?**], by which we can obtain a consistent parser from a pretty-printer.
- We give a surface language such that a pretty-printer written in it can be converted to a linear and treeless form by deforestation [**?**] which is suitable for inversion [**?**].
- We implemented our idea as a program transformation tool that generates parsers in Haskell[3]

In the sequel, we will give an overview of FliPpr (**??**) and then formally present the semantics and inversion firstly of a core language (**??**) and then a surface language (**??**) for FliPpr. After that we show how different pretty-printer combinators can be incorporated into FliPpr (**??**), and present a larger programming example (**??**). Finally, we discuss possible extensions (**??**) and related work (**??**), and conclude in **??**.

A preliminary version of this paper appeared in ESOP'13 [**?**]. The major difference to the preliminary version is **??**, which extends the back-end of FliPpr by adding more pretty-printing combinators. Also, **??** in addition discusses the applicability of FliPpr to other types of pretty-printing libraries, specifically Hughes' [**?**] and Bernardy's [**?**] that follow a different principle from Wadler's. In addition, **??** contains more discussions, especially on program inversion.

2 Overview

In this section, we present an overview of our technique using the subtraction language from the introduction as the running example.

2.1 Overall Architecture

Figure **??** shows the overall picture of FliPpr. A user of our system programs a pretty-printer in a surface language FliPpr, which is translated to a core language FliPpr Core that can be inverted. The example pretty-printer for the subtraction language is simple enough not

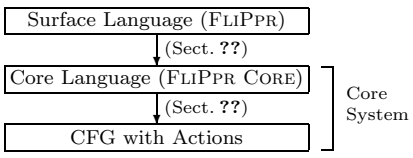

Fig. 1 Architecture of FliPpr

[3] Available at `https://bitbucket.org/kztk/flippr/`

to require any advanced features that the surface language provides, and the translation from the surface language to the core language is the identity operation in this case. Therefore, we focus on the core system in this section and postpone the discussion of the surface language to Sect. **??**.

2.2 Introducing Ugliness

Let us revisit the pretty-printer *ppr* defined in the previous section. If the function is inverted as it is, we can hope for no more than a parser that only recognizes pretty strings. This is neither the fault of function *ppr* nor of the inverter: a pretty-printer *ppr* (correctly) produces only pretty layouts, and an inverter cannot invent information that is not already carried by the function to be inverted. To remedy this information mismatch, we instrument the pretty-printer with additional information about non-pretty but nevertheless valid layouts.

Reinterpretation of ***line****.* A common source of prettiness is the clever interpretation of ***line***s either as a single space or a nicely indented new line depending on the environment. This effect can be simply eliminated by reinterpreting ***line*** as one or more whitespaces. Using this new interpretation in the derivation of a parser enables us to parse certain non-pretty layouts. For example, now the inverse of the pretty-printer can parse the following strings.

```
1   - 1   or   1
               - 1
```

These strings do not satisfy our notion of prettiness defined by *ppr*, and will not be produced by the pretty-printer, but will be accepted by the generated parser through the reinterpretation of ***line***s. Also note that this reinterpretation also means that we can safely ignore ***group*** and ***nest*** during inversion, because their sole purpose is to affect the behavior of ***line***s.

Still, this solution alone is not enough. Strings like `1 -  1` and `(1)-((1))` remain unparsable: the pretty-printer has dictated that there is only a single space between the operator and the second operand by using ***text*** `" "` instead of ***line***, and that there shouldn't be redundant parentheses. We need to find a way to alter these behaviors in parsing without losing pretty-printing.

Biased Choice. To annotate pretty-printers with information about non-pretty layouts, we introduce the choice operator `<?`. In pretty-printing the operator behaves as e_1 `<?` $e_2 = e_1$, ignoring the non-pretty alternative e_2; in parser derivation the operator is interpreted as a nondeterministic choice, which accepts both branches. The operator `<?` binds looser than `<>` and has the following algebraic properties.

Associativity	e_1 `<?` $(e_2$ `<?` $e_3) = (e_1$ `<?` $e_2)$ `<?` e_3
Distributivity-L	$(e_1$ `<?` $e_2)$ `<>` $e_3 = e_1$ `<>` e_3 `<?` e_2 `<>` e_3
Distributivity-R	e_1 `<>` $(e_2$ `<?` $e_2) = e_1$ `<>` e_2 `<?` e_1 `<>` e_3

For example, one can define variants of (white)spaces with the choice operator as follows.

```
nil   = text "" <? space                  -- zero-or-more spaces in parsing
space = (text " " <? text "\n") <> nil    -- one-or-more spaces in parsing
```

Here, *nil* and *space* pretty-print `""` and `" "` respectively, but represent zero-or-more and one-or-more whitespaces in parsing. We can now refactor our pretty-printer *ppr* with the aim of obtaining more robust parsers.

```
ppr x = ppr_ x <? text "(" <> nil <> ppr x <> nil <> text ")"
ppr_ One          = text "1"
ppr_ (Sub e1 e2) = group (ppr e1 <> nest 2 (lineN <> text "-" <> spaceN <> pprP e2))
pprP x = pprP_ x <? text "(" <> nil <> pprP x <> nil <> text ")"
pprP_ One          = text "1"
pprP_ (Sub e1 e2) =
    text "(" <> group (ppr e1 <> nest 2 (lineN <> text "-" <> spaceN <> pprP e2)) <> text ")"
spaceN = space <? text ""-- A variant of space that works as nil in parsing
lineN  = line <? text ""  -- A variant of line that works as nil in parsing
```

Note that we have separated the original definitions of *ppr* and *pprP* into two parts: the top level definitions introduce annotations for optional parentheses, and the actual pretty-printing is handled by worker functions that are subscripted. Optional whitespaces are also introduced by replacing *text* `" "` and *line* with *spaceN* and *lineN* respectively in the definitions.

This refactoring is semantic preserving with respect to pretty-printing, and at the same time brings in necessary information for robust parsing. For example, we can now expect the inverse program to parse strings like `1 - 1`, `(1)- ((1))`, and `(1 - (1))` correctly.[4]

2.3 Construction of CFG with Actions

So far, we have discussed how a user can provide a refactored pretty-printer that behaves like the original, but with additional information for non-pretty strings embedded. Our system FliPpr further transforms the program by removing the layouting and replacing `<?` with a nondeterministic choice ? to create an ugly-printer solely for inversion.

```
ppr x = ppr_ x ? "(" ++ nil ++ ppr x ++ nil ++ ")"
ppr_ One          = "1"
ppr_ (Sub e1 e2) = ppr e1 ++ line' ++ "-" ++ space' ++ pprP e2
...
```

We postpone a detailed discussion of the transformation to Sect. **??**. For now, it is sufficient to know that the above program nondeterministically produces a string that is valid for parsing, but not necessarily pretty.

[4] To also make strings like `" 1-1"` parsable, we can add a declaration $f\ x = nil \mathbin{<>} ppr\ x \mathbin{<>} nil$. However this addition does not give any new insight, and is omitted for simplicity.

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{rule}_1; \ldots; \mathit{rule}_n \\
\mathit{rule} & ::= & f\ p_1\ \ldots\ p_n = e \\
p & ::= & x \mid \mathsf{C}\ p_1\ \ldots\ p_n \\
e & ::= & \mathsf{text}\ s \mid e_1 \mathbin{<\!\!>} e_2 \mid \mathsf{line} \mid \mathsf{nest}\ n\ e \mid \mathsf{group}\ e \quad \text{(Wadler's Combinators)} \\
 & \mid & e_1 \mathbin{<?} e_2 \quad \text{(Biased Choice)} \\
 & \mid & f\ x_1\ \ldots\ x_n \quad \text{(Treeless Call)}
\end{array}
$$

Fig. 2 Syntax of FliPpr Core: f ranges over function, C ranges over constructors, x and x_is range over variables, s ranges over strings and n ranges over natural numbers.

Then, using our previous work on grammar-based inversion [**?**], the program can be inverted to construct the following grammar with actions (simplified for presentation).

$$
\begin{array}{lll}
\mathit{Ppr} \to \mathit{Ppr\_} & & \{\$1\} \\
\quad \mid \texttt{"("}\ \mathit{Nil}\ \mathit{Ppr}\ \mathit{Nil}\ \texttt{")"} & & \{\$3\} \\
\mathit{Ppr\_} \to \texttt{1} & & \{\mathsf{One}\} \\
\quad \mid \mathit{Ppr}\ \mathit{Line}'\ \texttt{"-"}\ \mathit{Space}'\ \mathit{PprP} & & \{\mathsf{Sub}\ \$1\ \$5\} \\
\ldots & &
\end{array}
$$

The correctness of the parser construction comes from our previous work [**?**].

Since FliPpr produces a CFG with actions, users have the choice of using any parser generator as long as it supports CFG. Although many existing parser generators and parsing combinators only support some subclasses of CFG such as LALR(1) and LL(1), algorithms (such as GLR [**?**], GLL [**?**], and derivative-based parsing [**?**]) and tools (such as GNU bison[5], happy[6], and Elkhound [**?**]) that support CFG are becoming popular now. Although those algorithms are generally slower than LALR(1) parsers for LALR(1) grammars, it has been reported that the overhead is only about 10% for a certain LALR(1) grammar in Elkhound thanks to its hybrid algorithm [**?**]. In our implementation, we use Frost *et al.* [**?**]'s top-down parser because of its simplicity.

3 Core Language and Parser Construction

In this section, we give the formal definition of the core language of FliPpr, FliPpr Core, and discuss parser construction by program inversion.

3.1 Syntax and Semantics

?? shows the syntax of FliPpr Core, a first-order functional language similar to one found in the introduction. We include Wadler's pretty-printing combinators [**?**] and the biased choice as primitive operators, and place two restrictions for later inversion:

– Function calls must be *treeless* [**?**]: they take only variables as arguments.

[5] `https://www.gnu.org/software/bison/`

[6] `https://www.haskell.org/happy/`

$$\frac{\exists (f\ \widetilde{p} = e).\ \widetilde{p}\sigma' = \widetilde{x}\sigma \quad \sigma' \vdash e \Downarrow v}{\sigma \vdash f\ \widetilde{x} \Downarrow v} \qquad \frac{\sigma \vdash e_1 \Downarrow v_1}{\sigma \vdash e_1\ \texttt{<?}\ e_2 \Downarrow v_1} \qquad \frac{}{\sigma \vdash \mathit{text}\ s \Downarrow \mathit{text}\ s}$$

$$\frac{\{\sigma \vdash e_i \Downarrow v_i\}_{i=1,2}}{\sigma \vdash e_1 \mathbin{<>} e_2 \Downarrow v_1 \mathbin{<>} v_2} \qquad \frac{}{\sigma \vdash \mathit{line} \Downarrow \mathit{line}} \qquad \frac{\sigma \vdash e \Downarrow v}{\sigma \vdash \mathit{nest}\ n\ e \Downarrow \mathit{nest}\ n\ v} \qquad \frac{\sigma \vdash e \Downarrow v}{\sigma \vdash \mathit{group}\ e \Downarrow \mathit{group}\ v}$$

Fig. 3 The call-by-value pretty-printing semantics of FliPpr Core.

$$\frac{\exists (f\ \widetilde{p} = e).\ \widetilde{p}\sigma' = \widetilde{x}\sigma \quad \sigma' \vdash e \Downarrow_{\mathrm{ND}} s}{\sigma \vdash f\ \widetilde{x} \Downarrow_{\mathrm{ND}} s} \qquad \frac{\sigma \vdash e_i \Downarrow_{\mathrm{ND}} s_i}{\sigma \vdash e_1\ \texttt{<?}\ e_2 \Downarrow_{\mathrm{ND}} s_i}\ i = 1,2 \qquad \frac{}{\sigma \vdash \mathit{text}\ s \Downarrow_{\mathrm{ND}} s}$$

$$\frac{\{\sigma \vdash e_i \Downarrow_{\mathrm{ND}} s_i\}_{i=1,2}}{\sigma \vdash e_1 \mathbin{<>} e_2 \Downarrow_{\mathrm{ND}} s_1 +\!\!+ s_2} \qquad \frac{s \in \mathit{White}^+}{\sigma \vdash \mathit{line} \Downarrow_{\mathrm{ND}} s} \qquad \frac{\sigma \vdash e \Downarrow_{\mathrm{ND}} s}{\sigma \vdash \mathit{nest}\ n\ e \Downarrow_{\mathrm{ND}} s} \qquad \frac{\sigma \vdash e \Downarrow_{\mathrm{ND}} s}{\sigma \vdash \mathit{group}\ e \Downarrow_{\mathrm{ND}} s}$$

Fig. 4 Nondeterministic printing semantics of FliPpr Core.

– Variable use must be *linear*: every bound variable in a rule is used *exactly once* on the right-hand side. An exception is with `<?`. For e_1 `<?` e_2, the two branches are supposed to be both linear. Thus, they contain the same set of free variables. For example, assuming f is linear, then $g\ x = f\ x$ `<?` $f\ x$ is linear, but $h\ x = \mathit{line}$ `<?` $f\ x$ and $k\ x = \mathit{line}$ `<?` *text* `"s"` are not.

For simplicity, we often omit the rule separator ";" if no confusion would arise. We use vector notation $\widetilde{x}$ for a sequence $x_1, \ldots, x_n$, and abuse the notation to write $f\ \widetilde{x}$ for $f\ x_1\ \ldots\ x_n$.

The formal pretty-printing semantics of the language is shown in Fig. **??**. We write $\sigma \vdash e \Downarrow v$ if under environment σ, expression e evaluates to value v. Values are closed expressions that only consist of Wadler's combinators (*i.e.*, we do not evaluate Wadler's combinators). The environment σ is a mapping from variables to terms (*i.e.*, expressions or patterns). We write $t\sigma$ for the term obtained from t by replacing free variables x in t with $\sigma(x)$. Notice that $\widetilde{x}\sigma$ in the first rule represents the actual arguments of f; recall that a variable is not an expression in this language. Pattern matching is nondeterministic in this semantics.

We do not define formally the semantics of Wadler's combinators, as our discussion in this paper does not depend on it. However, we define the reinterpretation of the combinators and the biased choice `<?` for parser generation (firstly mentioned in Sect. **??**, where *line*s are seen as one-or-more whitespaces and `<?` as a true nondeterministic choice). As shown in Fig. **??**, the reinterpretation is defined similarly to the pretty-printing semantics; the main difference is that it returns a string nondeterministically, pretty or not. We write $\sigma \vdash e \Downarrow_{\mathrm{ND}} s$ if, under the environment σ, e nondeterministically evaluates to a string s. Here, $+\!\!+$ is string concatenation, and White^+ is the set of non-empty strings that consist only of `" "` and `"\n"`. That is, $\mathit{White}^+ = \bigcup_{1 \le i} S_i$ where S_i is defined inductively by: $S_1 = \{\texttt{" "}, \texttt{"\n"}\}$ and $S_{n+1} = \{s_1 +\!\!+ s_2 \mid s_1 \in S_1, s_2 \in S_n\}$. The possible evaluation results of the nondeterministic semantics, which covers both pretty and non-pretty strings, is a super set of what Wadler's combinators may produce if evaluated in the original semantics. Thanks to treelessness and linearity, the sets of strings de-

fined by $L_e = \{s \mid \exists \sigma.\, \sigma \vdash e \Downarrow_{\mathrm{ND}} s\}$ for expressions e are exactly those that are expressible by CFGs. This fact enables us to use CFG-parsers for inverses, which will be shown in the following. Also note that due to linearity, call-by-value and call-by-name coincide for the language, even with nondeterminism (assuming that Wadler's combinators and string operations are strict). This is handy later when we require a call-by-value semantics for program inversion [**?**], and a call-by-name semantics for fusion [**?**] in the surface language (Sect. **??**).

3.2 Parser Construction by Inversion

To invert programs written in the core language, we firstly perform a semantic-preserving transformation to remove the pretty-printing combinators, and obtain a syntax that is recognizable by our grammar-based inversion system [**?**].

3.2.1 Converting to Nondeterministic Programs

This step is done by "forgetting smart layouting mechanism", through the following rewriting rules.

$$\begin{array}{lll} \mathit{text}\ s \longrightarrow s & \mathit{group}\ e \longrightarrow e & e_1 \mathbin{<\!>} e_2 \longrightarrow e_1 +\!\!+ e_2 \\ \mathit{nest}\ n\ e \longrightarrow e & \mathit{line} \longrightarrow \mathit{space} & e_1 \mathbin{<?} e_2 \longrightarrow e_1 \,?\, e_2 \end{array}$$

Here, *space* is a rewritten version (according to the rules above) of its definition in Sect. **??** defined as the following (the operator ? is nondeterministic choice).

$$\mathit{space} = (\texttt{" "} \,?\, \texttt{"\textbackslash n"}) +\!\!+ \mathit{nil} \qquad \mathit{nil} = \texttt{""} \,?\, \mathit{space}$$

The formal semantics of the obtained nondeterministic programs is defined straightforwardly by adding the following rules.

$$\frac{}{\sigma \vdash s \Downarrow s} \qquad \frac{\sigma \vdash e_i \Downarrow v}{\sigma \vdash e_1 \,?\, e_2 \Downarrow v}\ i = 1, 2 \qquad \frac{\{\sigma \vdash e_i \Downarrow v_i\}_{i=1,2}}{\sigma \vdash e_1 +\!\!+ e_2 \Downarrow v_1 +\!\!+ v_2}$$

The behaviors of s, ? and $+\!\!+$ are the same as the reinterpretations of *text* s, `<?` and `<>`, respectively; we use different symbols to clarify that the conversion discards the pretty-printing semantics. Note that due to linearity and treelessness, the call-time (or call-by-value or IO) choice and the run-time (or call-by-name or OI) choice [**?**] coincide.

We write $\underline{f}$ and $\underline{e}$ as the rewritten version of f and e. The following lemma states that the rewriting is semantic preserving.

Lemma 1 (Semantic Preservation) $\sigma \vdash e \Downarrow_{\mathrm{ND}} s$ *iff* $\sigma \vdash \underline{e} \Downarrow s$. □

Rules of F_f. For function $\underline{f}$, we generate:

$$\begin{array}{lll} F_f \to E_{e_1}\ \{\textbf{let}\ \sigma = \$1\ \textbf{in}\ (\widetilde{p}_1)\sigma\} & & \\ \quad \ldots & & \text{if } \underline{f} \text{ has rules } \underline{f}\ \widetilde{p} = \underline{e_1}; \ldots; \underline{f}\ \widetilde{p}_n = \underline{e_n}. \\ \quad \mid\ E_{e_n}\ \{\textbf{let}\ \sigma = \$1\ \textbf{in}\ (\widetilde{p}_n)\sigma\} & & \end{array}$$

Rules of E_e. For expression $\underline{e}$, we generate:

$$\begin{array}{lll} E_e \to F_f\ \left\{\begin{array}{l}\textbf{let}\ (t_1, \ldots, t_n) = \$1 \\ \textbf{in}\ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}\end{array}\right\} & & \text{if } \underline{e} = \underline{f}\ x_1\ \ldots\ x_n \\ E_e \to E_{e_1}\ E_{e_2}\quad \{\$1 \uplus \$2\} & & \text{if } \underline{e} = \underline{e_1} +\!\!+ \underline{e_2} \\ E_e \to s\quad \{\emptyset\} & & \text{if } \underline{e} = s \\ E_e \to E_{e_1}\quad \{\$1\} & & \\ \quad \mid\ E_{e_2}\quad \{\$1\} & & \text{if } \underline{e} = \underline{e} = \underline{e_1}\ ?\ \underline{e_2} \end{array}$$

Here, $\uplus$ merges two environments assuming that their domains are disjoint. Note that this disjoint property is guaranteed by linearity.

Fig. 5 Construction of CFG with actions.

3.2.2 Grammar-Based Inversion

Grammar-based inversion [**?**] is an inversion method that reduces the problem of finding x such that $f(x) = y$ for a given y to that of parsing based on a grammar corresponding to the set $\{y \mid \exists x.y = f(x)\}$ together with transformations on the parse trees. The original paper [**?**] mainly discussed regular tree grammars [**?**]. This paper, instead, considers CFG as a variant.

Specifically, we convert the rewritten program to a context-free grammar with actions[7] that computes the inverse of the rewritten program. The basic idea of this grammar construction is to read a rule of a program as a production rule of a grammar, and to use semantic actions to track how variables (*i.e.*, inputs) are passed.

In the inversion, we construct two sorts of non-terminals: F_f for functions $\underline{f}$ and E_e for expressions $\underline{e}$. For a function $\underline{f}$ that takes $t_1, \ldots, t_n$ and returns s, F_f is used to parse string s, and the semantic action returns the original inputs $(t_1, \ldots, t_n)$. For an expression $\underline{e}$ such that $\sigma \vdash \underline{e} \Downarrow s$, E_e is used to parse string s, and the semantic action returns the original environment σ. The generation of the production rules and semantics actions are presented in Fig. **??**. The grammar in Sect. **??** is a simplified version of the grammar obtained by this generation.

We write $[\![N]\!]_{\mathrm{P}}(s)$ for the set of results returned by the semantics actions, when s is parsed with start symbol N (the subscript P means "parse"). The following lemma holds.

Lemma 2 (Correctness of Inversion)

- $\sigma \vdash \underline{e} \Downarrow s$ *and* $\mathsf{dom}(\sigma) = \mathsf{fv}(\underline{e})$ *iff* $\sigma \in [\![E_e]\!]_{\mathrm{P}}(s)$,
- $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \vdash \underline{f}\ x_1\ \ldots\ x_n \Downarrow s$ *iff* $(t_1, \ldots, t_n) \in [\![F_f]\!]_{\mathrm{P}}(s)$.

Proof Similar to [**?**]. □

[7] As a slight difference, the original paper [**?**] uses transformations on parse trees (or more precisely, derivation trees of productions), instead of semantic actions.

Let *ppr* be a single-argument function defined in FliPpr Core, and *parse* be a function defined by $parse\ s = [\![F_{ppr}]\!]_{\mathrm{P}}(s)$. Then, the following theorem is a special case of the above lemma.

Theorem 1 $\{x \mapsto t\} \vdash ppr\ x \Downarrow_{\mathrm{ND}} s$ *iff* $t \in parse\ s$. □

The set *parse s* contains at most one element if *ppr* is injective. Note that the inversion can produce arbitrary CFGs, and this is the reason why FliPpr requires parser generators that support CFG without restriction.

4 Surface Language: Making it More Flexible

The core language, FliPpr Core, is restricted to be linear and treeless, which is expressive enough for CFG parsing, but may be cumbersome to program in at times. In this section, we present a surface language name FliPpr that has a relaxed form of the restrictions, and through fusion techniques (specifically deforestation [**?**] or supercompilation [**?**]), programs written in the surface language are transformed to treeless and linear programs in the core language.

4.1 Problems with Programming in the Core Language

Let us consider extending the subtraction language with division and variables.

data $E = \cdots \mid \mathsf{Div}\ E\ E \mid \mathsf{Var}\ String$

Recall that we used two mutually recursive functions *ppr* and *pprP* to control bracketing issues around "-". In general, when there are many operators with different precedence levels, it suffices to use a function for each precedence level. For example, assuming "-" has precedence-level 6 and "/" has precedence-level 7 as they do in Haskell, a pretty-printer can be written as follows.

```
ppr x =  ppr_5 x      -- 5 is the lowest precedence level
...
ppr_-5 (Sub x y) = ... ppr_5 x ... text "-" ... ppr_6 y ...      -- (1)
ppr_-5 (Div x y) = ... ppr_6 x ... text "/" ... ppr_7 y ...      -- (2)
...
ppr_-6 (Sub x y) = text "(" <> nil <> ... {- the RHS of (1) -} ... <> nil <> text ")"
ppr_-6 (Div x y) = ... {- the RHS of (2) -} ...
...
ppr_-7 (Sub x y) = text "(" <> nil <> ... {- the RHS of (1) -} ... <> nil <> text ")"
ppr_-7 (Div x y) = text "(" <> nil <> ... {- the RHS of (2) -} ... <> nil <> text ")"
```

The various auxiliary functions are subscripted by numbers which are the precedence levels of the contexts the expressions are being printed. For example in ppr_{-6}, Sub is printed with added parentheses as its precedence-level 6 is not higher than the context; on the other hand, there is no parentheses for Div as its precedence-level 7 is higher than the context's level 6. Astute readers may have noticed that there are a lot of repetitions in the above definition largely due to the treeless restriction.

Another problem that it is non-trivial to separate variable names from predefined names. For example, let us consider pretty-printing for Var x. One may be tempted to write $ppr\_\ (\mathsf{Var}\ x) = \mathit{text}\ x$ but a parser derived from the above will parse "`-`" as Var `"-"`, because there is no information in the above definition that specifies valid variable names. We can improve the pretty-printer as follows.

$$\begin{array}{ll} ppr\_\ (\mathsf{Var}\ x) = f\ x & g\ [\,] \quad\ = \mathit{text}\ \texttt{""} \\ f\ (\texttt{'a'} : x) = \mathit{text}\ \texttt{"a"} \diamond g\ x & g\ (\texttt{'a'} : x) = \mathit{text}\ \texttt{"a"} \diamond g\ x \\ \ldots & \ldots \\ f\ (\texttt{'z'} : x) = \mathit{text}\ \texttt{"z"} \diamond g\ x & g\ (\texttt{'z'} : x) = \mathit{text}\ \texttt{"z"} \diamond g\ x \end{array}$$

Note that strings are represented as lists of characters. This function $ppr\_$ is intentionally partial (undefined for Var `"-"`). We have successfully restricted variable names to range over lower-case alphabets, but in a cumbersome way.

4.2 An Overview

To reduce the programming effort, we propose a surface language FliPpr, which has relaxed linearity and treelessness restrictions, and is equipped with a shorthand notation for expressing name ranges. In this language, a pretty-printer for the extended subtraction language can be written as follows.

$$\begin{array}{l} ppr\ x = go\ 5\ x \\ go\ i\ x = manyPars\ (go\_\ i\ x) \\ go\_\ i\ \mathsf{One} \quad\ = \mathit{text}\ \texttt{"1"} \\ go\_\ i\ (\mathsf{Var}\ x) \quad = \mathit{text}\ x\ \mathbf{as}\ \texttt{[a-z]+} \\ go\_\ i\ (\mathsf{Sub}\ e_1\ e_2) = \\ \quad parIf\ (i \geq 6)\ (\mathit{group}\ (go\ 5\ e_1 \diamond \mathit{nest}\ 2\ (\mathit{line}' \diamond \mathit{text}\ \texttt{"-"} \diamond \mathit{space}' \diamond go\ 6\ e_2))) \\ go\_\ i\ (\mathsf{Div}\ e_1\ e_2) = \\ \quad parIf\ (i \geq 7)\ (\mathit{group}\ (go\ 6\ e_1 \diamond \mathit{nest}\ 2\ (\mathit{line}' \diamond \mathit{text}\ \texttt{"/"} \diamond \mathit{space}' \diamond go\ 7\ e_2))) \end{array}$$

Here, *manyPars* and *parIf* are defined as:

$$\begin{array}{l} parIf\ b\ d = \mathbf{if}\ b\ \mathbf{then}\ par\ d\ \mathbf{else}\ d \\ manyPars\ d = d \mathbin{<?} par\ (manyPars\ d) \\ par\ d = \mathit{text}\ \texttt{"("} \diamond \mathit{nil} \diamond d \diamond \mathit{nil} \diamond \mathit{text}\ \texttt{")"} \end{array}$$

This program differs from the one in the core language in the following ways:

1. The auxiliary functions *manyPars*, *parIf* and *par* are used and applied to non-variable arguments, enabling users to avoid duplicating frequently-occurring patterns such as $\mathit{text}\ \texttt{"("} \diamond \mathit{nil} \diamond \cdots \diamond \mathit{nil} \diamond \mathit{text}\ \texttt{")"}$.
2. Instead of embedding precedence-levels into function names, we pass them as arguments and inspect them by **if** and $\leq$ for bracketing. (These were previously impossible due to the linearity and treelessness restrictions.)
3. A new construct $\mathit{text}\ x\ \mathbf{as}\ r$ is used to avoid explicit recursion on strings.

Item 3 of the above is rather easy to deal with. For Item 1, we borrow the idea of program fusion [**?**, **?**, **?**] to make sure that these auxiliary functions are fused away. For Item 2, we use partial evaluation to erase statically-computable arguments such as precedence-levels. The statically-computable arguments are separated from the rest through types.

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{rule}_1 \ldots \mathit{rule}_n \\
\mathit{rule} & ::= & f\ p_1\ \ldots\ p_n = e \\
e & ::= & \mathit{text}\ s \mid e_1 \mathbin{<\!\!>} e_2 \mid \mathit{line} \mid \mathit{nest}\ n\ e \mid \mathit{group}\ e \mid e_1 \mathbin{<\!?\!>} e_2 \quad \text{(Combinators)} \\
 & \mid & \mathit{text}\ x\ \mathbf{as}\ r \quad \text{(Annotated Text)} \\
 & \mid & x \quad \text{(Variable)} \\
 & \mid & f\ e_1\ \ldots\ e_n \quad \text{(Call)} \\
 & \mid & \mathbf{if}\ \mathit{pred}\ e_1\ \ldots\ e_n\ \mathbf{then}\ e_\mathrm{t}\ \mathbf{else}\ e_\mathrm{f} \quad \text{(Static Branching)} \\
 & \mid & c \quad \text{(Constant)} \\
p & ::= & x \mid \mathsf{C}\ p_1\ \ldots\ p_n \\
c & ::= & \ldots \text{ any constants } \ldots \\
r & ::= & \ldots \text{ regular expression } \ldots
\end{array}
$$

Fig. 6 Syntax of FliPpr: *pred* are Boolean predicates.

$$\boxed{\Theta, \Gamma, \Delta \vdash e : \tau}$$

$$\frac{}{\Theta, \Gamma, \{x : \tau\} \vdash x : \tau} \quad \frac{}{\Theta, \Gamma, \emptyset \vdash x : \Gamma(x)} \quad \frac{}{\Theta, \Gamma, \emptyset \vdash c : \mathsf{St}}$$

$$\frac{\Theta, \Gamma, \Delta \vdash e : \mathsf{Doc}}{\Theta, \Gamma, \Delta \vdash \mathit{nest}\ n\ e : \mathsf{Doc}} \quad \frac{\{\Theta, \Gamma, \Delta_i \vdash e_i : \mathsf{Doc}\}_{1 \le i \le n} \quad op = \mathit{text}\ \texttt{"s"}, \mathit{group}, (\mathbin{<\!\!>}), \mathit{line}}{\Theta, \Gamma, \biguplus_{1 \le i \le n} \Delta_i \vdash op\ e_1\ \ldots\ e_n : \mathsf{Doc}}$$

$$\frac{\{\Theta, \Gamma, \Delta \vdash e_i : \mathsf{Doc}\}_{i=1,2}}{\Theta, \Gamma, \Delta \vdash e_1 \mathbin{<\!?\!>} e_2 : \mathsf{Doc}} \quad \frac{}{\Theta, \Gamma, \{x : \mathsf{AST}\} \vdash \mathit{text}\ x\ \mathbf{as}\ r : \mathsf{Doc}}$$

$$\frac{\{\Theta, \Gamma, \emptyset \vdash e_i : \mathsf{St}\}_{1 \le i \le n} \quad \{\Theta, \Gamma, \Delta \vdash e_b : \tau\}_{b=\mathrm{t,f}}}{\Theta, \Gamma, \Delta \vdash \mathbf{if}\ \mathit{pred}\ e_1\ \ldots\ e_n\ \mathbf{then}\ e_\mathrm{t}\ \mathbf{else}\ e_\mathrm{f} : \tau}$$

$$\frac{\{\Theta, \Gamma, \Delta_i \vdash e_i : \tau_i\}_{1 \le i \le n} \quad \Theta(f) = \tau_1 \to \cdots \to \tau_n \to \mathsf{Doc}}{\Theta, \Gamma, \biguplus_{1 \le i \le n} \Delta_i \vdash f\ e_1\ \ldots\ e_n : \mathsf{Doc}}$$

$$\boxed{\Theta \vdash f\ p_1\ \ldots\ p_n = e}$$

$$\frac{\begin{array}{c}\Theta(f) = \tau_1 \to \cdots \to \tau_n \to \mathsf{Doc} \\ \exists \Gamma, \Delta_1, \ldots, \Delta_n \quad \{\Gamma, \Delta_i \vdash p_i : \tau_i\}_{1 \le i \le n} \quad \mathsf{dom}(\Gamma) \subseteq \biguplus_{1 \le i \le n} \mathsf{fv}(p_i) \\ \Theta, \Gamma, \biguplus_{1 \le i \le n} \Delta_i \vdash e : \mathsf{Doc}\end{array}}{\Theta \vdash f\ p_1\ \ldots\ p_n = e}$$

$$\boxed{\Gamma, \Delta \vdash p : \tau}$$

$$\frac{\Gamma(x) = \mathsf{St}}{\Gamma, \emptyset \vdash x : \mathsf{St}} \quad \frac{\tau \in \{\mathsf{AST}, \mathsf{Doc}\}}{\Gamma, \{x : \tau\} \vdash x : \tau} \quad \frac{\{\Gamma, \Delta_i \vdash p_i : \tau\}_{1 \le i \le n} \quad \tau \in \{\mathsf{AST}, \mathsf{St}\}}{\Gamma, \biguplus_{1 \le i \le n} \Delta_i \vdash \mathsf{C}\ p_1\ \ldots\ p_n : \tau}$$

Fig. 7 Typing rules: here $\uplus$ represents disjoint union.

4.3 Surface Language: FliPpr

Figure **??** shows the syntax of the surface language FliPpr. The treeless restriction is replaced by a relaxed one that will be discussed towards the end of this subsection. The language has constants as expressions, such as the precedence levels of operators found in the previous subsection. Used as arguments, such constants can be eliminated at compilation time through partial evaluation; we call such constants *static information*. The **if** branchings inspect static information, and are eliminable statically as well.

We use a type system to distinguish static information (of type St) from other kinds of values such as the input ASTs (of type AST) and the pretty-printing results (of type Doc). The type system ensures that static information are eliminable through (offline) partial-evaluation [**?**], and variable uses are

linear. Formally, primitive types τ and function types σ are defined by:

$$\tau ::= \mathsf{AST} \mid \mathsf{St} \mid \mathsf{Doc} \qquad \sigma ::= \tau_1 \to \cdots \to \tau_n \to \tau$$

Typing judgment $\Theta, \Gamma, \Delta \vdash e : \tau$ reads that under function-type environment Θ, non-linear type environment Γ and linear type environment Δ, e has type τ. Similarly, we define $\Gamma, \Delta \vdash p : \tau$ and $\Theta \vdash f\ p_1\ \ldots\ p_n = e$ for patterns and declarations. Figure **??** shows the typing rules, which are mostly self-explanatory. Notably, the uses of variables of type AST and Doc have to be linear, as dictated by the rules. The linearity restriction of AST variables is inherited from the core language, while that of Doc variables is required for the correctness of fusion; it is known that the deforestation is not correct for non-linear *and* non-deterministic programs [**?**]. A program is assumed to have a distinguished entry point function of type $\mathsf{AST} \to \mathsf{Doc}$. The type Doc is treated as a black box in the language; nothing except Wadler's combinators can handle Doc data. Only variables can have type AST.

Treeless Restriction We replace the universal treeless restriction of the core language by a typed one: only arguments of type AST or Doc are restricted to be variables. Moreover, we view programs in FliPpr as *multi-tier* systems [**?**]: every function is associated to a natural number called *tier*, and every function call occurring in the body of a tier-i function must be to a tier-j ($\leq i$) function. Tiers of functions are easily inferred by topological sorting of the call-graph. A program is called *tiered-treeless* if for every call of a tier-k function f occurring in the body of a tier-k function, the arguments (of type AST or Doc) passed to the call must be variables. The pretty-printer defined in Sect. **??** is tiered-treeless: functions *ppr*, *go* and *go_* belong to tier 3, function *manyPars* belongs to tier 2, and other functions belong to tier 1.

We omit a formal semantics of FliPpr, as it is a straightforward extension of FliPpr Core. Similar to the case of FliPpr Core, the evaluation results of the call-by-value and the call-by-name semantics coincide in the surface language due to linearity.

4.4 Conversion to FliPpr Core

FliPpr is elaborated to FliPpr Core through a number of program transformations: (1) desugaring expressions of the form *text* x **as** r, (2) partial-evaluating static information, (3) fusing higher-tier functions. Steps (1) and (2) above are straightforward adaptation of existing technologies, while step (3) is new and uses a property specific to our surface language. In what follows, we discuss the steps one by one.

4.4.1 Desugaring *text* x **as** r

We firstly convert r to a deterministic automaton. Then, we replace *text* x **as** r with $f_{q_0}\ x$ where q_0 is an initial state of the automaton, and, for each state q,

a function f_q is defined as follows: function f_q has a rule f_q (`'a'` : x) = $f_{q'}$ x if the automaton has a transition rule $(q, \mathtt{a}, q')$, and has a rule f_q [ ] = *text* `""` if q is a final state of the automaton. For the example in Sect. **??**, the regular expression `[a-z]+` can be expressed in a deterministic automaton with two states, and the functions f and g correspond to the two states.

4.4.2 Partial-Evaluating St*-Expressions*

A role of our type system is to perform binding-time analysis [**?**]; the expressions of type St can be statically evaluated, assuming that predicate applications are terminating. Thus, standard offline partial evaluation (see [**?**]) suffices to eliminate all the St-expressions and thus we omit the details. For the example in Sect. **??**, we obtain the partially evaluated functions as below.

$$
\begin{array}{ll}
ppr\ x = go_5\ x & go_{\_6}\ (\mathsf{Sub}\ x\ y) = \ldots go_5\ x \ldots go_6\ y \ldots \\
\ldots & go_{\_6}\ (\mathsf{Div}\ x\ y) = par\ (\ldots go_6\ x \ldots go_7\ y \ldots) \\
go_{\_5}\ (\mathsf{Sub}\ x\ y) = \ldots go_5\ x \ldots go_6\ y \ldots & \ldots \\
go_{\_5}\ (\mathsf{Div}\ x\ y) = \ldots go_6\ x \ldots go_7\ y \ldots & go_{\_7}\ (\mathsf{Sub}\ x\ y) = par\ (\ldots go_5\ x \ldots go_6\ y \ldots) \\
\ldots & go_{\_7}\ (\mathsf{Div}\ x\ y) = par\ (\ldots go_6\ x \ldots go_7\ y \ldots)
\end{array}
$$

Roughly speaking, thanks to the type AST → Doc of the entry point function, the type system guarantees that every St-type expression must be a constant itself or a part of some constant obtained by pattern-matching, and thus can eliminated by partial-evaluation.

4.4.3 Fusing Functions to Obtain 1*-tier Programs*

We show the transformation of 2-tiered programs to 1-tiered programs, with the understanding that the procedure can be applied iteratively to transform m-tiered programs to 1-tiered programs.

The transformation is done by deforestation [**?**]. Roughly speaking, deforestation (or, supercompilation [**?**][8]) performs call-by-name evaluation of expressions; but instead of computing a value, it produces a new expression that has the same behavior as the original one but with intermediate data structures eliminated. Without loss of generality, we assume that AST arguments appear before Doc arguments in function calls. The deforestation procedure $\mathcal{D}[\![e]\!]$ is defined as follows.

- $\mathcal{D}[\![op\ e_1\ \ldots\ e_n]\!] = op\ \mathcal{D}[\![e_1]\!]\ \ldots\ \mathcal{D}[\![e_n]\!]$, where op ranges over *text* `"s"`, (<>), *line*, *nest* i, *group* and (<?).

[8] Because of the linearity, Wadler's deforestation [**?**] and (positive) supercompilation [**?**] coincide for the surface language FliPpr. Also notice that we assume non-deterministic pattern-matching and thus need not convey negative information in the deforestation process. Thus, generalized partial computation (GPC) [**?**, **?**] also coincide with the above two for FliPpr [**?**].

– $\mathcal{D}[\![f\ \widetilde{x}\ \widetilde{e}]\!] = f_{\widetilde{e}}\ \widetilde{x}\ \widetilde{z}$. Assuming $\widetilde{x}$ have type AST (recall that only variables have type AST), $\widetilde{e}$ have type Doc, and $\{\widetilde{z}\}$ are the free variables in $\widetilde{e}$, the newly generated function $f_{\widetilde{e}}$ is defined as $f_{\widetilde{e}}\ \widetilde{p}\ \widetilde{z} = \mathcal{D}[\![e[\widetilde{y} \mapsto \widetilde{e}]]\!]$ for each corresponding rule $f\ \widetilde{p}\ \widetilde{y} = e$ in the definition of f (with proper α-renaming). Here, we do not repeatedly generate rules of $f_{\widetilde{e}}$ if they are already generated (up to renaming of the free variables in $\widetilde{e}$).

The above procedure follows from the original one [**?**], and is simplified to suit the restricted surface language. The procedure terminates if the number of functions $f_{\widetilde{e}}$ generated in the latter case is finite. By using $\mathcal{D}[\![e]\!]$, we replace every tier-2 rule $f\ \widetilde{p}\ \widetilde{y} = e$ with $f\ \widetilde{p}\ \widetilde{y} = \mathcal{D}[\![e]\!]$.

Example 1 We deforest the pretty-printer defined in Sect. **??**.

The tier-2 function *manyPars* is transformed into the following.

$$
\begin{aligned}
&\mathit{manyPars}\ d &&= d\ \texttt{<?}\ \mathit{par}_{\mathit{manyPars}\ d}\ d\\
&\mathit{par}_{\mathit{manyPars}\ d}\ d &&= \mathsf{text}\ \texttt{"("}\ \texttt{<>}\ \mathit{nil}\ \texttt{<>}\ \mathit{manyPars}\ d\ \texttt{<>}\ \mathit{nil}\ \texttt{<>}\ \mathsf{text}\ \texttt{")"}
\end{aligned}
$$

And iteratively, we can now apply the procedure to the function go_5 (reproduced below), which is in tier-2 after the above transformation.

$$\mathit{go}_5\ x = \mathit{manyPars}\ (\mathit{go}_{-5}\ x)$$

After renaming $\mathit{par}_{\mathit{manyPars}\ d}$ to *parMP*, we have the following tier-1 functions

$$
\begin{aligned}
&\mathit{go}_5\ x &&= \mathit{manyPars}_{\mathit{go}_{-5}\ x}\ x\\
&\mathit{manyPars}_{\mathit{go}_{-5}\ x}\ x &&= \mathit{go}_{-5}\ x\ \texttt{<?}\ \mathit{parMP}_{\mathit{go}_{-5}\ x}\ x\\
&\mathit{parMP}_{\mathit{go}_{-5}\ x}\ x &&= \mathsf{text}\ \texttt{"("}\ \texttt{<>}\ \mathit{nil}\ \texttt{<>}\ \mathit{go}_{-5}\ x\ \texttt{<>}\ \mathit{nil}\ \texttt{<>}\ \mathsf{text}\ \texttt{")"}
\end{aligned}
$$

assuming calls $go_5\ x$ are transformed too. This is similar to inlining except that the deforestation handles recursive functions such as *manyPars*. □

Note that in the deforestation process, we treat Wadler's combinators as constructors because Doc-values are black boxes. This is key to termination; if we allow pattern-matching on Doc-values, then we can make a tiered-treeless program for which deforestation runs infinitely. As a result, Theorem **??** can be generalized and $\mathcal{D}[\![e]\!]$ terminates for tier-n expression e.

Theorem 2 (Termination) *For tier-2 expression e, $\mathcal{D}[\![e]\!]$ terminates.*

Proof (Sketch) All expressions $\widetilde{e}$ in $\mathcal{D}[\![f\ \widetilde{x}\ \widetilde{e}]\!]$ must be tier-2 expressions in the original program or just variables, which implies the finiteness of the number of functions $f_{\widetilde{e}}$ generated in the deforestation process. □

Theorem 3 *The resulting tier-1 program is treeless and linear.* □

The correctness of the deforestation is known for call-by-name languages [**?**]. Note that call-by-value and call-by-name coincide in our surface language.

It is worth remarking that deforestation (supercompliation, or generalized partial computation [**?**,**?**]) is known to be an extension of online partial evaluation [**?**]; and online partial evaluation that classifies static and dynamic inputs

in the transformation time, is more powerful than offline partial evaluation that classifies static and dynamic inputs ahead of the transformation [**?**]. This suggests that we could remove the step (2) above on partial evaluation. Nevertheless, we prefer to keep the separate step with the binding-time analysis by types, so that the set of input programs for FliPpr is made clear by types, and deforestation process may focus on handling multi-tier programs.

5 Back-end Extensions: Enriching Combinators

Extensions to Wadler's pretty-printer combinators [**?**] can be incorporated into FliPpr. In this section, we demonstrate the inclusion of additional combinators together with some post-processing for the language.

5.1 Primitive Combinators

Leijen extended Wadler's combinators [**?**] with a number of primitives (namely *linebreak*, *align*, *fill* and *fillBreak*) in the library `wl-pprint`[9]. They can be included in FliPpr. Specifically, the syntax of expressions both in the core and the surface languages is extended as:

$$e ::= \cdots \mid \mathit{linebreak} \mid \mathit{align}\ e \mid \mathit{fill}\ n\ e \mid \mathit{fillBreak}\ n\ e$$

In what follows, together with brief explanations of their behaviors, we show how the combinatiors are translated to nondeterministic programs in parser construction.

linebreak is a variant of *line*; it represents a newline with indentation similar to *line*, but when placed under *group* it may be rendered as the empty string instead of a single space. This would be useful for rendering punctuation symbols such as opening braces "`{`" for grouping statements.

The translation of this combinator is straightforward.

$$\mathit{linebreak} \longrightarrow \mathit{nil}$$

where *nil* nondeterministically prints zero-or-more spaces (Sect. **??**). Recall that *line* is converted to *space*.

align adjusts the current indentation level like *nest* n, but it sets the current indentation level to the column number of the current rendering position, instead of a fixed n. This combinator is key to realize relative indentation, like Hughes' or Bernardy's combinators [**?**, **?**].

Let us consider rose trees as an example.

data *RTree* = Node *String* [*RTree*]

[9] `https://hackage.haskell.org/package/wl-pprint`

$$
\begin{aligned}
&\mathit{pprRT}\ (\mathsf{Node}\ n\ \mathit{ts}) = \mathit{text}\ n\ \mathbf{as}\ \texttt{[a-z]+} \mathbin{<>} \mathit{spaceN} \mathbin{<>} \mathit{text}\ \texttt{"["} \mathbin{<>} \mathit{group}\ (\mathit{align}\ (\\
&\qquad\qquad \mathit{nil} \mathbin{<>} \mathit{pprRTs}\ \mathit{ts} \mathbin{<>} \mathit{nil} \mathbin{<>} \mathit{text}\ \texttt{"]"}))\\
&\mathit{pprRTs}\ [\,] \quad\ = \mathit{text}\ \texttt{""}\\
&\mathit{pprRTs}\ (t : \mathit{ts}) = \mathit{pprRTs}'\ t\ \mathit{ts}\\
&\mathit{pprRTs}'\ t\ [\,] \quad\ = \mathit{pprRT}\ t\\
&\mathit{pprRTs}'\ t\ (t' : \mathit{ts}) = \mathit{pprRT}\ t \mathbin{<>} \mathit{nil} \mathbin{<>} \mathit{text}\ \texttt{","} \mathbin{<>} \mathit{lineN} \mathbin{<>} \mathit{pprRTs}'\ t'\ \mathit{ts}
\end{aligned}
$$

For example, *ppr* (Node `"apple"` [Node `"orange"`, Node `"banana"`]) will be rendered as

```
apple [orange, banana]    or    apple [orange,
                                       banana]
```

depending on the screen width. Notice that the indentation of "`banana`" is relative to the length of its parent label (*i.e.*, "`apple`").

The translation of this combinator is similar to that of *nest* n:

$$\mathit{align}\ e \longrightarrow e$$

fill and fillBreak The expression *fill* n e inserts spaces after e to make it n-wide if e is narrower than n, and otherwise returns e; *fillBreak* is similar to *fill*, but behaves as e <> *nest* n *linebreak* if e's wider than n. They can be used for lists of bindings such as maps, to align the positions of "`=`" for example.

We can translate these combinator by appending *nil*.

$$\mathit{fill}\ n\ e \longrightarrow e +\!\!+ \mathit{nil} \qquad \mathit{fillBreak}\ n\ e \longrightarrow e +\!\!+ \mathit{nil}$$

In the implementation in `wl-print`, the rendered width of e can be negative as it is calculated as the difference of the horizontal position before and after rendering e, and thus *fill* n and *fillBreak* n can insert spaces even when n is negative. This is why we insert *nil* in the translation regardless of n.

5.2 Derived Combinators for Spacing and Post-Processing

Recall that, in addition to the usual pretty-printing, FliPpr programs specify where non-pretty spaces are allowed for effective parsing (Sect. **??**). However, our experience shows that directly inserting *nil* or *space* for this purpose is error-prone. We have observed that it is rather rare to have concatenation ("<>") without allowing additional spaces; that is, "<>" is usually used together with spacing expressions such as *nil*, *space* and *line*. This observation leads to the design of the following combinators.

$$
\begin{aligned}
x \mathbin{<\!\!>} y &= x \mathbin{<>} \mathit{nil} \mathbin{<>} y & \mathit{spacesAround}\ d &= \mathit{nil} \mathbin{<>} d \mathbin{<>} \mathit{nil}\\
x \mathbin{<\!\!\!>} y &= x \mathbin{<>} \mathit{space} \mathbin{<>} y & x \mathbin{</>} y &= x \mathbin{<>} \mathit{line} \mathbin{<>} y\\
x \mathbin{<\!\!\!>_N} y &= x \mathbin{<>} \mathit{spaceN} \mathbin{<>} y & x \mathbin{</>_N} y &= x \mathbin{<>} \mathit{lineN} \mathbin{<>} y
\end{aligned}
$$

For example, *pprRT* in Sect. **??** can be rewritten as below with the new combinators (one might notice that this version differs from the previous one

by printing *nil* before, instead of after, *group* and *align*, but this makes no difference in either pretty-printing nor parsing).

```
pprRT (Node n ts) = text n as [a-z]+ <+>_N text "[" <> group (align (
                      pprRTs ts <> text "]"))
...
pprRTs' t (t' : ts) = pprRT t <> text "," </>_N pprRTs' t' ts
```

The new combinators simply the definition, and are effective in avoiding errors related to missing spacing expressions.

5.2.1 An Issue: Unnecessary Ambiguity

A caveat with the new combinators is that programmers are now more likely to write concatenations of spacing expressions, which result in ambiguous grammar.

For example, let us consider the rose-tree example again, but instead of relative indentation, we insert a *linebreak* with *nest*ing after printing `"["`.

```
pprRT (Node n ts) = text n as [a-z]+ <+>_N text "[" <> group (nest 2 (
                      linebreak <> pprRTs ts <> text "]"))
```

Notice that "*align*" is replaced with "*nest* 2" and "*linebreak* <> " is inserted before "*pprRTs ts*". From the definition, we obtain the following ambiguous grammar (after some inlining of nonterminals for readability).

$$\begin{array}{ll} \mathit{PprRT} \to \ldots \texttt{"["}\ \mathit{Nil}\ \mathit{Nil}\ \mathit{Nil}\ \ldots & \mathit{Nil} \to \texttt{""} \mid \mathit{Space} \\ & \mathit{Space} \to S\ \mathit{Nil} \end{array}$$

Here, the nonterminal S is assumed to generate single spaces (*e.g.*, `" "` and `"\n"`). Recall that *linebreak* is translated to *nil* for parser construction. This grammar is ambiguous due to the consecutive occurrences of *Nil*; Nil^n can produce a k-sequence of "␣" in $\mathrm{O}(k^{(n-1)})$ ways.

This ambiguity is caused by the abstraction offered by the new combinators. Instead of going back to the old ways of writing <> and explicit spacings, we use post-processing to eliminate the ambiguity.

5.2.2 Post-Processing

We employ post-processing to eliminate ambiguity caused by the concatenation of spacing expressions (especially, *nil* and *space*). This is addressed by a common technique in the formal language theory.

Let S and S^* be terminals. Intuitively, they represent a single and zero-or-more space(s) respectively, but are frozen [**?**] to be atomic *terminals* for manipulation. Then we can convert spacing expressions using S and S^* (for example, *nil* and *linebreak* are converted to S^*, and *line* and *space* to $S\,S^*$).

Our task now is to eliminate the ambiguity caused by concatenation of S and S^* when they are thawed to nonterminals (*i.e.*, S^* has productions

$S^* \to S\,S^* \mid$ `""`, and S produces a single space). This is done by canonizing consecutive occurrences of S and S^* in strings generated by a grammar, leveraging the equations $S^*S^* = S^*$ and $SS^* = S^*S$. For example, SS^*SS^* is canonized to SSS^*.

This canonization can be implemented by the following transducer, presented as a Haskell-like program traversing a list of terminals.

$$
\begin{array}{lll}
q_0\ [\,] & = [\,] & q_1\ [\,] = S : [\,] \\
q_0\ (S : x) & = q_1\ x & q_1\ (S : x) = S : q_1\ x \\
q_0\ (S^* : x) & = q_2\ x & q_1\ (S^* : x) = S : q_2\ x \\
q_0\ (a : x) & = a : q_2\ x & q_1\ (a : x) = S : a : q_0\ x
\end{array}
\qquad
\begin{array}{lll}
q_2\ [\,] & = S^* : [\,] \\
q_2\ (S : x) & = S : q_2\ x \\
q_2\ (S^* : x) & = q_2\ x \\
q_2\ (a : x) & = S^* : a : q_0\ x
\end{array}
$$

Here a denotes a terminal except S or S^*, *i.e.*, a character. Intuitively q_0, q_1 and q_2 are the states to traverse strings after ordinary characters, S and S^* respectively. The states q_2 and q_3 produce S and S^* as late as possible, in order to avoid producing multiple S^*s in a sequence of S and S^*.

Now we fuse the transducer and the original grammar to obtain a new grammar in which S^* occurs at most once in every sequence of S and S^* in the generated strings. This is done by fusing a (linear) top-down tree transducer to a macro tree transducer [**?**], viewing a grammar as a transformation from parse trees (not as semantic action results) to strings represented as cons-list. We do not go into the details of this fusion, but briefly explain the fusion methods on the generated grammar in Fig. **??**. The idea is that, for each nonterminal N in the original grammar, we prepare nonterminals N^{ij} $(i, j \in \{0, 1, 2\})$ that generates a string w' if and only if N generates w and $q_i\ (w +\!\!+ x) = w' +\!\!+ q_j\ x$. The nonterminals N^{ij} have productions $N^{ij} \to \alpha_k^{ij}\ \{A\}$ if there is a production $N \to \alpha\ \{A\}$ in the original grammar, where $\{\alpha_1^{ij}, \ldots, \alpha_n^{ij}\} = \mathit{conv}^{ij}(\alpha)$ for each $i, j \in \{0, 1, 2\}$. The function conv^{ij} is defined as:

$$
\begin{array}{ll}
\mathit{conv}^{ij}(s) & = \begin{cases} \{s'\} & \text{if } q_i(s) = s' +\!\!+ q_j\ [\,] \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{conv}^{ij}(N) & = \{N^{ij}\} \\
\mathit{conv}^{ij}(N_1\ N_2) & = \{N_1^{ik}\ N_2^{kj} \mid k \in \{0, 1, 2\}\}
\end{array}
$$

Notice that we assumed an input grammar that is constructed as in Fig. **??**; thus a right-hand side of a production is either a string, a single non-terminal or a concatenation of two nonterminals. After this generation, for the start symbol N_0, we generate the start symbol N_0' that has the following productions

$$N_0' \to N_0^{00}\ \ \{\$1\} \mid N_0^{01}\ S\ \ \{\$1\} \mid N_0^{02}\ S^*\ \ \{\$1\}$$

together with the obvious rules for S and S^*. The resulting grammar can be at most nine-times bigger than the original one. But since most of the nonterminals do not produce any strings, removing rules involving such nonterminals would reduce the grammar size effectively.

The fusion is harmless; it does not introduce any additional ambiguity. This is because q_0 only affects the sequence of S and S^* and leaves other characters untouched, while keeping the "semantics" (*i.e.*, the set of produced

strings) of the S/S^*-sequences. For example, suppose that the sets of strings produced from N_0^{01} and $N_0^{02}\ S^*$ overlap. We first state that, when S and S^* are "frozen" to be terminals, the sets of strings produced from N_0^{00} and $N_0^{02}\ S^*$ are disjoint, as the former set contains only the empty string and strings ended with a normal character while the latter contains only strings ended with S^*. This means that the inverse images of these sets for q_0 are disjoint, and thus the original grammar has different ways to produce a string from the former set and a string from the latter set. Recall that q_0 keeps the semantics of S and S^*, and thus having an overlap between N_0^{00} and $N_0^{02}\ S^*$ after thawing S and S^* to nonterminals means that the inverse images overlap after thawing, which indicates the ambiguity of the original grammar. In general, assuming that a string has two parse trees, we consider their "intermediate" parse trees where S and S^* are not expanded. Then, we can safely assume that the intermediate parse trees are different as q_0 removed ambiguity caused by S/S^* sequences, and the rest of the discussion is similar to the above.

Although the fusion removes ambiguity caused by *concatenation* of *nil* and *space*, it leaves some ambiguity due to spaces. For example, the result is ambiguous if we use *space*′ and *line*′ more than once in space sequences, where both will be converted to a nonterminal N with the rules $N \to$ `""` $\mid S\,S^*$. The concatenation $N\,N$ produces "␣" in two ways, but this ambiguity cannot be removed by q_0 because it comes from the different choices in the production of two Ns instead of concatenation of S and S^*. For example, when the start symbol has a production rule $N_0 \to N\,N$, the fusion generates obviously ambiguous productions $N_0' \to$ `""` $\mid S\,S^* \mid S\,S^* \mid S\,S\,S^*$ (after inlining). An easy workaround is to normalize $N \to$ `""` $\mid S\,S^*$ to $N \to S^*$ before the fusion. Another remedy is to treat *space*′ and *line*′ as primitives that will be converted to S^* for parser construction.

It is worth noting that this post-interpretation of S (and S^*) has another advantage. So far, the translation of some combinators such as *line* is hardwired. As programming languages usually have syntax for comments, one way to handle them in FliPpr is to treat comments as single spaces so that *line* will be able to skip them in parsing just like spacing characters. Post-interpretation of S enables this adjustment to the interpretation of *line*.

6 A Larger Example

In the introduction, we advertised that *"we, and many others who read this paper, will not need to do it [writing both parser and pretty-printer] for their own language implementations."*. In this section, we demonstrate the feasibility of this goal by writing a pretty-printer for FliPpr Core in FliPpr, which, if fed to the FliPpr system, will generate a parser for the core language.

The ASTs of the core language can be expressed by the following datatype.

```
type Prog = [Rule]
data Rule = Rule String [Pat] Exp
data Exp  = ECon String [Exp] | EOp Op Exp Exp | EVar String [Exp]
```

```
data Pat  = PVar String | PCon String [Pat]
data Op   = OCat | OAlt      -- <> and <?
```

We leave out *nest* and *text* for simplicity. In the datatype, we use EVar both for variables and function calls to avoid ambiguity in grammars.

The overall principle of our pretty-printing is to insert breaks after `=`, and before `<>` and `<+`, with 2-space indentation. We start with lists of rules, and insert separators with optional whitespaces *nil* `<>` *text* `";"` `<>` $line'$ between individual rules.

```
ppr x = spacesAround (pprRules x)

pprRules []       = text ""
pprRules (r : rs) = pRules r rs

pRules r' []       = pprRule r'
pRules r' (r : rs) = pprRule r' <> text ";" </>_N pRules r rs
```

For each rule, its right-hand side may start a new line.

```
pprRule (Rule f ps e) =
    group (var f <+> pprPats ps <> text "=" <> nest 2 (line' <> pprExp e))
var x = text x as [a-z][_a-zA-Z0-9']*
```

A list of patterns is treated in a similar but simpler way to a list of rules.

```
pprPats []       = text ""
pprPats (p : ps) = pprPat p <+> pprPats ps
```

Redundant parentheses in patterns are admissible for the generated parser, but will not be produced by the pretty-printer.

```
pprPat p = manyPars (pprPat_ p)

pprPat_ (PVar x)          = var x
pprPat_ (PCon c [])       = con c
pprPat_ (PCon c (p : ps)) = par (con c <+> pPats p ps)

con f = text x as [A-Z][_a-zA-Z0-9']*
```

Expressions are printed according to the precedence-levels and associativities of the operators.

```
pprExp e = go 4 e

go i e = manyPars (go_ i e)

go_ i (ECon c [])       = con c
go_ i (ECon c (e : es)) = parIf (i ≥ 9) (con c <+> pExps e es)
go_ i (EOp OAlt e1 e2) =
    parIf (i ≥ 5) (group (go 5 e1 <> nest 2 (line' <> text "<+" <+>_N go 4 e2)))
go_ i (EOp OCat e1 e2) =
    parIf (i ≥ 6) (group (go 6 e1 <> nest 2 (line' <> text "<>" <+>_N go 5 e2)))
go_ i (EVar f [])       = var f
go_ i (EVar f (e : es)) = parIf (i ≥ 9) (var f <+> pExps e es)
```

Finally, a list of expressions is printed in a similar way to a list of patterns.

```
pExps e' []       = go 9 e'
pExps e' (e : es) = go 9 e' <+> pExps e es
```

7 Discussion

We discuss aspects of FliPpr including possible extensions and limitations.

Non-Structured Values in AST ASTs may contain non-structured values such as `Int`. It is easy to extend the core system to handle the issue. For example, our implementation supports the syntax *text* f x **as** r where f is a bijection between a non-structured value and a string representation of it. The bijections can be read bidirectionally for either pretty-printing or parsing.

Higher-Order Functions Higher-order functions, such as `map` and `foldr` are useful in writing pretty-printers. For example, *pprPats* and *pprExps* in **??** can be conveniently implemented by `foldr`. However, general use of higher-order functions in pretty-printing may produce grammars that go beyond CFG. The syntactic linearity restriction has to be lifted, as most of the higher-order functions use the functional arguments more than once on the right-hand sides. We may need to consult some linear type system.

In line with the spirit of the surface language, a way forward is to use higher-order functions only when they can be fused away. A sufficient condition for fusion is the absence of λ-abstractions and partial-applications. Along the line, the authors recently, based on staged-computation, proposed a programming language that allows one to write bidirectional [**?**]/invertible transformations in a similar style to unidirectional transformations together with higher-order functions [**?**]. We leave the combining of the idea to FliPpr as future work.

Spacing We have demonstrated that careful use of whitespaces in the definition of the pretty-printer is an effective way to control the behavior of the generated parser. For example, for pretty-printing constructor application in **??**, we wrote (*con* c ⇿ *pExps* e *es*); the use of ⇿ (concatenation with one-or-more whitespaces) allows us to parse "`S Z`" or "`S  Z`" as valid strings. However, it is difficult to express the use of spaces that are dynamically dependent on the printing results of adjacent expressions, especially with nondeterminism. In the above example, if we were to know that the argument of the application is printed in parentheses as "`(Z)`", then in some syntax the space between the constructor and the argument can be omitted as in "`S(Z)`". On the other hand, we cannot simply replace *space* with *space*$'$, because we don't want to accept "`SZ`" as a valid constructor application. One possible solution to the problem is to try to extend the generate parsers with a lexing phase. However, it may require some major surgery to the current system.

Non-Linearity In the literature of tree transducers [**?**], the discussion of linearity can be separated into input- and output-linearity. In our case, variables of AST type can be seen as inputs, and those of Doc type as outputs.

Regarding the non-linearity of AST-typed variables, sometimes we want to pretty-print the same AST twice; for example, an element e in XML is

printed as <e>...</e>. A naive solution to admit this behavior is to check the equivalence of values of duplicated variables in semantic actions. More concretely, we relax $\uplus$ to allow overlapping domains in the operands, and define $\{x \mapsto v\} \uplus \{x \mapsto v\} = \{x \mapsto v\}$. This naive solution works effectively for XML, because the number of possible ASTs is usually finite. However, in general parsing becomes undecidable with non-linear use of AST variables, as shown in [**?**] (Theorem 4.4). Thus, for this kind of non-linear uses, a method that checks the finiteness of parse trees is required.

The non-linearity of Doc values has non-trivial interaction with nondeterminism. Without linearity, the call-by-value and the call-by-name semantics may cease to coincide. This is a problem because call-by-value is suitable for grammar-based inversion [**?**], but call-by-name is suitable for deforestation [**?**]. We also need to resort to grammars beyond CFGs, which may pose difficulties in inversion. It is a challenging problem to find a sweet spot between obtaining efficient inverses and supporting fusion in the surface language.

Pretty-Printing Combinators Other Than Wadler's Other than Wadler's combinators and extensions that we have seen, Hughes' pretty-printing combinators [**?**], which have a different design, are also popular among Haskell programmers. More recently, Bernardy [**?**] developed a pretty-printing combinators for slower but prettier printing. These libraries follow a different principle from Wadler's. Especially, there are multiple combinators for handling layouting (*i.e.*, inserting spaces or newlines).

One of the key combinators in these libraries is <|>; by writing d_1 <|> d_2, we can choose the prettier result from d_1 and d_2 depending on different criteria [**?**, **?**]. However, this is not the only place that affects layouting. The combinator $$ vertically stacks the layouts: $d_1 \,\$\$\, d_2$ places d_2 beneath d_1. This placement is relative in the sense that d_2 starts at the same column with d_1. For example, by writing *text* " " <> $(d_1 \,\$\$\, d_2)$, both d_1 and d_2 are indented. Unlike Wadler's, the nesting combinator *nest* n directly affects layouting in different ways in these libraries. In Hughes', *nest* k d indents d by k, but its effect can be canceled by concatenation, as d' <> *nest* k $d = d'$ <> d. In Bernardy's, *nest* n is a derived combinator defined by *nest* n d = *text* (*replicate* n " ") <> d, but expected to be used only in the function *hang* n x $y = (x$<>$y)$<|>$(x$$$*nest* k $y)$.

Taking these behavors into account, a possible reinterpretation of these combinators in nondeterministic (ugly) printing is as follows.

$$e_1 \texttt{<|>} e_2 \longrightarrow e_1 \,?\, e_2 \qquad e_1 \,\$\$\, e_2 \longrightarrow e_1 +\!\!+ \mathit{space} +\!\!+ e_2 \qquad \mathit{nest}\ n\ e \longrightarrow \mathit{nil} +\!\!+ e$$

However, the above conversions usually lead to ambiguous grammars when derived combinators involving <|> are used, for example *sep* $[d_1, \ldots, d_n] = (d_1$ <> *text* " " <> $\cdots$ <> *text* " " <> $d_n)$ <|> $(d_1$ $$ $\cdots$ $$ $d_n)$. To avoid such ambiguity, derived combinators will have to be reinterpreted individually.

Indentation Sensitivity Recall that the FliPpr produces CFGs with actions, which puts the limit on the obtained parser. For example, FliPpr cannot produce parsers for indentation sensitive languages such as Haskell and Python.

It is worth noting that the CFG restriction can actually be circumvented in some indentation sensitive languages. For example, GHC, a Haskell compiler, and Python deal with indentation at the lexing phase by inserting special tokens based on indentation levels. This suggests that having a separate lexing phase may enable FliPpr to handle indentation sensitive languages.

Another approach to solve the issue is to use more elaborated interpretation of *nest* to produce indentation-sensitive extension of context-free grammars [**?**]. This would also require major surgery to our system.

8 Related Work

Different approaches have been proposed to simultaneously derive a parser and a printer from some intermediate descriptions. In particularly, one could start from an annotated CFG specification to derive both a parser and a pretty-printer [**?**]. Compared to these systems, FliPpr offers finer control over pretty-printing. In particular, we are able to deal with contextual information and to define auxiliary functions like *par* in printing, which is made conveniently available by the surface language. Other approaches include invertible syntax descriptions [**?**] based on invertible programming, and BNFC-meta [**?**] based on meta programming. Both work recognizes the importance of good printing, but is not able to support pretty-printing.

Another approach to guarantee the correctness (in the sense of **??**) of a parser and pretty-printer pair is to develop the pair with a correctness proof by certified programming. In Danielsson [**?**]'s framework in Agda, a user writes a grammar and a pretty-printer, but the correctness of the pretty-printer with respect to the grammar is guaranteed by construction with the help of dependent types, as his pretty-printing combinators convey proofs together. Since his framework targets coinductive monadic parsers, it can handle more expressive grammars beyond CFG. However, as the grammar and its pretty-printer are programmed separately, there is a maintenance problem: a grammar change may trigger non-trivial change to the pretty-printer.

There are also general-purpose bidirectional/invertible languages [**?**, **?**, **?**, **?**, **?**] that in theory can be used to build the printer/parser pair from the definition of one of them. Notably quotient lenses [**?**] are designed to include a representative of a quotient before performing bidirectional conversions; in our case, roughly speaking this quotient operation is the erasure of redundant whitespaces and parentheses. However, there is a gap between the theoretical possibility and practical execution. In particular, the pretty-printing libraries of Wadler [**?**] and Hughes [**?**] are not only user-friendly but also highly optimized. Moreover, for efficient parsing we have to perform whole-program analysis (as in conventional parsing algorithms like LR-k) or use sophisticated data structures and memoization [**?**, **?**]. It is not obvious how these sophisticated implementations can be packed into a bidirectional program. In our approach, we avoid this problem by using grammar-based inversion [**?**], which

generates grammars and outsources the parsing algorithms to selected parser generators.

It is, however, worth mentioning that using bidirectional transformations [**?**] has an advantage that FliPpr does not have. Since bidirectional transformations allow one data to contain some information that the other data does not have and keep the information as possible in updating, we can make a printer/parser pair where the printer keeps the original syntactic sugars and layout information as possible when AST changes [**?**, **?**], which is useful for refactoring and converting the core language's evaluation steps to the sugared language's evaluation steps for debugging. In addition to transformations between strings to ASTs, bidirectional transformations between several artifacts such as concrete parse trees and tokens with or without layout information have several benefits including visual editing, code formatting and layout-preserving transformations [**?**]. It is an interesting future direction how to make FliPpr bidirectional while keeping its efficiency.

FliPpr uses a variant of grammar-based inversion [**?**] to produce parsers represented in CFG with actions. This is not the only research using formal grammars in program inversion or inverse computation.[10] Yellin [**?**] discusses inversion of a class of string-to-string attribute grammars [**?**]. The method can be seen as an extension of synchronous grammars [**?**]—transformations by two CFGs that share the same parse tree module permutation of children. XSugar [**?**] is also based on synchronous grammars, but it is more designated for transformations between XML and plain text representations. Glück and Kawabe [**?**] use LR(0) parsing method for post-processing of the inverted program so that they can obtain more deterministic inverse programs also for accumulative tail-recursive functions. Matsuda *et al.* [**?**, **?**] adopt the idea of grammar-based inversion for polynomial-time inverse computation of a class of functional programs that can have multiple data traversals and accumulative computations, namely parameter-linear macro-tree transducers [**?**]. The method uses a variant of the Cocke-Younger-Kasami (CYK) algorithm for trees. Although the grammars are implicit in the method, if restricted to monadic trees, they can be seen as a non-linear extension of linear context-free rewriting systems [**?**] and multiple context-free grammars [**?**]. Recently, Hu and D'Antoni [**?**] proposed an inversion method for symbolic transducers. Symbolic transducers, unlike the usual transducers that transit states based on one input symbol, transit states based on predicates for a fixed number of symbols. This enables one to describe transformations on lists of which elements comes from an infinite domain, or even for finite cases reduces the number of states of a transducer. Thanks to the advantage, although its recursion structures are restricted compared with any of the above, the method

[10] Strictly speaking, program inversion and inverse computation are different notions; the former is a transformation to yield an inverse program while the latter is an interpretation method to run a program in an inverse way. The difference between the two become rather unimportant for grammar-based inversion that produces grammars with additional transformations, as whether we transform (as parser generators) or interpret (as the most of parser combinators) the grammars is left for the later process.

can effectively handle practical examples such as Base64 encoding and utf-8 encoding.

There are a lot of discussions on how to make deforestation (supercompilation) terminating (*e.g.*, [**?**]) for Turing-complete languages. These approaches use conditions to give up fusion, and reuse the already-generated deforested functions. As a result, these approaches may fail to fuse some functions, and thus are not suitable for our purpose. The completeness of deforestation, in the sense whether all nested calls are fused away, has not been the focus of study in the literature. Notable exceptions are Wadler's original work [**?**] and tree transducer fusion [**?**, **?**, **?**]. However, there is a gap between treeless functions and tree transducers; especially, treeless functions can take multiple inputs. It is not obvious how existing results can be directly applied in our case.

In the design of FliPpr, we used deforestation/supercompilation to have a more user-friendly syntax (FliPpr) for the invertible core (FliPpr Core). The connection between supercompilation and program inversion is also found in other contexts. There is an inverse computation method called the universal resolving algorithm [**?**] which is based on driving [**?**] in supercompilation. A similar idea is known as needed narrowing [**?**], which is now served as a basis of the functional logic programming language Curry [**?**].

9 Conclusion

In this paper, we proposed a method to derive parsers from pretty-printers. We start with a program written in a language equipped with Wadler's pretty-printing combinators [**?**], and an additional "choice" operator. The choice operator allows us to enrich the pretty-printer with information about valid but yet non-pretty strings, without changing the pretty-printing behavior. This enriched pretty-printer can be transformed and inverted using grammar-based inversion [**?**] to produce a CFG parser. For the inversion to be possible, the language is restricted to be linear and treeless [**?**]. We also provide a surface language that has relaxed restrictions, which eases programming. The surface language is transformed into the linear and treeless language through fusion.

We feel that the specific problem we addressed in this paper has much wider implications. It suggests a general framework for program inversion problems with "information mismatch". A compression/decompression pair is another example of this kind. For example in runlength encoding, we want to decode both `A3B1` and `A1A2B1` as `AAAB`, but an encoder "prefers" the former. It is an interesting problem to see how our technique may apply in these contexts.