FliPpr: A Prettier Invertible Printing System

Kazutaka Matsuda[1] and Meng Wang[2]

[1] The University of Tokyo
[2] Chalmers University of Technology

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

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

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 to each other: very informally, parsing a pretty-printed program should succeed, and produces 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 [5].

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

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 101–120, 2013.

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 [2, 4, 17] . We have intentionally omitted the prefix "pretty-" from the mentioning of printers here because few of the existing work is actually producing pretty-printers in the sense of Hughes [10] and Wadler [22].[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.

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

Using the language we propose in this paper, which is based on Wadler's library [10], one can define a pretty-printer 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*. Roughly speaking, *text* s converts a string s to a layout, $d_1 \mathbin{<>} d_2$ is an infix binary operator that concatenates two layouts d_1 and d_2, which binds looser than prefix applications, and *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. (In this paper, we write "space" for the space character and write "whitespace" for the space character and the newline character. Other kinds of spaces such as horizontal tabs are not discussed as they do not yield new insight.)

The function *ppr* pretty-prints Sub (Sub One One) (Sub One One) as

```
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

[1] The Syn system [2] is capable of handling non-contextual layouts, which can be seen as a limited form of prettiness.

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, in this paper we propose a novel approach: the programmer provides a carefully turned pretty-printer (which is slightly annotated with some additional information for parsing), and our system invert it to obtain a consistent parser. We claim the following benefits of our approach:

- **Fine-Grained Control over Pretty-Printing**. Our language based on Wadler's library [22] 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 [22]. For parsing, we can use any parser generator that supports full CFG.

The technique of program inversion used in FliPpr is not new; it is a direct consequence of our previous work [15]. 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 [15], 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 [21] which is suitable for inversion [15].
- We implemented our idea as a program transformation tool that generates parsers in Haskell[2].

2 Overview

In this section, we present an overview of our technique using the subtraction language from the introduction as the running example. Figure 1 shows the overall picture of FliPpr. A user of our system programs a pretty-printer in a surface language, which is translated to a core language that can be inverted. The example pretty-printer for the subtraction language is simple enough not to require any advanced features that the surface

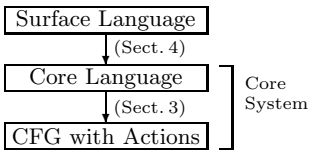

Fig. 1. Architecture of FliPpr

[2] Available at `http://www-kb.is.s.u-tokyo.ac.jp/~kztk/FliPpr/`

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. 4.

As a start, let's 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.

2.1 Introducing Ugliness

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 $e_1 \mathbin{<\!\!+} e_2 = e_1$, 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	$e_1 \mathbin{<\!\!+} (e_2 \mathbin{<\!\!+} e_3) = (e_1 \mathbin{<\!\!+} e_2) \mathbin{<\!\!+} e_3$
Distributivity-L	$(e_1 \mathbin{<\!\!+} e_2) \mathbin{<>} e_3 = e_1 \mathbin{<>} e_3 \mathbin{<\!\!+} e_2 \mathbin{<>} e_3$
Distributivity-R	$e_1 \mathbin{<>} (e_2 \mathbin{<\!\!+} e_2) = e_1 \mathbin{<>} e_2 \mathbin{<\!\!+} e_1 \mathbin{<>} e_3$

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

```
nil   = text "" <+ space                  -- (zero-or-more whitespaces in parsing)
space = (text " " <+ text "\n") <> nil    -- (one-or-more whitespaces 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 (line' <> text "-" <> space' <> pprP e2))
pprP x = pprP_ x <+ text "(" <> nil <> pprP x <> nil <> text ")"
pprP_ One         = text "1"
pprP_ (Sub e1 e2) =
    text "(" <> nil <> group (ppr e1 <> nest 2 (line'
                                <> text "-" <> space' <> pprP e2)) <> nil <> text ")"
space' = space <+ text "" -- (zero-or-more whitespaces in parsing)
line'  = line <+ text ""  -- (zero-or-more whitespaces 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 $space'$ and $line'$ 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.[3]

2.2 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. 3. For now, it is sufficient to know that the above program nondeterministically produces a string that is valid for parsing, but not necessarily pretty.

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

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

Fig. 2. Syntax of the core language: f ranges over function, C ranges over constructors, x and x_is range over variables and n range over natural numbers

Then, using our previous work on grammar-based inversion [15], 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 [15]. Since FliPpr produces a CFG with actions, users have the choice of using any parser generator that supports full CFG. In our implementation, we use Frost *et al.* [8]'s top-down parser.

3 Core Language and Parser Construction

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

3.1 Syntax and Semantics

Figure 2 shows the syntax of our core language, a first-order functional language similar to one found in the introduction. We include Wadler's pretty-printing combinators [22] and the biased choice as primitive operators, and place two restrictions for later inversion:

- Function calls must be *treeless* [21]: they take only variables as arguments.
- Variable use must be *linear*: every bound variable in a rule is used *exactly once* on the right-hand side. A notable exception is with $\mathbin{<\!\!+}$. For $e_1 \mathbin{<\!\!+} e_2$, 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 $g\ x = f\ x \mathbin{<\!\!+} f\ x$ is linear, but $h\ x = \mathit{line} \mathbin{<\!\!+} f\ x$ and $k\ x = \mathit{line} \mathbin{<\!\!+} \mathit{text}\ \texttt{"s"}$ are not.

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

$$\frac{\exists(f\ \widetilde{p} = e).\ \widetilde{p}\Gamma' = \widetilde{x}\Gamma \quad \Gamma' \vdash e \Downarrow v}{\Gamma \vdash f\ \widetilde{x} \Downarrow v} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1}{\Gamma \vdash e_1 \mathbin{<\!\!+} e_2 \Downarrow v_1}$$

$$\frac{}{\Gamma \vdash \mathit{text}\ \texttt{"s"} \Downarrow \mathit{text}\ \texttt{"s"}} \qquad \frac{\{\Gamma \vdash e_i \Downarrow v_i\}_{i=1,2}}{\Gamma \vdash e_1 \mathbin{<>} e_2 \Downarrow v_1 \mathbin{<>} v_2} \qquad \frac{}{\Gamma \vdash \mathit{line} \Downarrow \mathit{line}}$$

$$\frac{\Gamma \vdash e \Downarrow v}{\Gamma \vdash \mathit{nest}\ n\ e \Downarrow \mathit{nest}\ n\ v} \qquad \frac{\Gamma \vdash e \Downarrow v}{\Gamma \vdash \mathit{group}\ e \Downarrow \mathit{group}\ v}$$

Fig. 3. The call-by-value pretty-printing semantics of the language

$$\frac{\exists(f\ \widetilde{p} = e).\ \widetilde{p}\Gamma' = \widetilde{x}\Gamma \quad \Gamma' \vdash e \Downarrow_{\mathrm{ND}} s}{\Gamma \vdash f\ \widetilde{x} \Downarrow_{\mathrm{ND}} s} \qquad \frac{\Gamma \vdash e_i \Downarrow_{\mathrm{ND}} s_i}{\Gamma \vdash e_1 \mathbin{<\!\!+} e_2 \Downarrow_{\mathrm{ND}} s_i}\ i = 1, 2$$

$$\frac{}{\Gamma \vdash \mathit{text}\ \texttt{"s"} \Downarrow_{\mathrm{ND}} \texttt{"s"}} \qquad \frac{\{\Gamma \vdash e_i \Downarrow_{\mathrm{ND}} s_i\}_{i=1,2}}{\Gamma \vdash e_1 \mathbin{<>} e_2 \Downarrow_{\mathrm{ND}} s_1 +\!\!+ s_2} \qquad \frac{s \in \bigcup_{1 \le i} S_i}{\Gamma \vdash \mathit{line} \Downarrow_{\mathrm{ND}} s}$$

$$\frac{\Gamma \vdash e \Downarrow_{\mathrm{ND}} s}{\Gamma \vdash \mathit{nest}\ n\ e \Downarrow_{\mathrm{ND}} s} \qquad \frac{\Gamma \vdash e \Downarrow_{\mathrm{ND}} s}{\Gamma \vdash \mathit{group}\ e \Downarrow_{\mathrm{ND}} s}$$

Fig. 4. Nondeterministic printing semantics of the language

The formal pretty-printing semantics of the language is shown in Fig. 3. We write $\Gamma \vdash e \Downarrow v$ if under environment Γ, expression e evaluates to value v. Values are closed expressions that only consist of Wadler's combinators (*i.e.*, we don't evaluate Wadler's combinators). The environment Γ is a mapping from variables to terms (*i.e.*, expressions or patterns). We write $t\Gamma$ for the term obtained from t by replacing free variables x in t with $\Gamma(x)$. Pattern matching is nondeterministic in this semantics.

We do not define formally the semantics of Wadler's combinators, as our discussion in this paper is not dependent on it. However, we define the reinterpretation of the combinators and the biased choice <+ for parser generation, firstly mentioned in Sect. 2, where *line*s are seen as one-or-more whitespaces and <+ as a true nondeterministic choice. As shown in Fig. 4, 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 $\Gamma \vdash e \Downarrow_{\mathrm{ND}} s$ if, under the environment Γ, e nondeterministically evaluates to a string s. Here, S_i is the set of i-long consecutive whitespaces, inductively defined by: $S_1 = \{\texttt{" "}, \texttt{"\textbackslash n"}\}$ and $S_{n+1} = \{s_1 +\!\!+ s_2 \mid s_1 \in S_1, s_2 \in S_n\}$, and $+\!\!+$ is the concatenation of two strings. 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 defined by $L_e = \{s \mid \Gamma \vdash e \Downarrow_{\mathrm{ND}} s\}$ 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 rest of this section. 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 [15], and a call-by-name semantics for fusion [21] in the surface language (Sect. 4).

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 [15].

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

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

Here, *space* is a rewritten version (according to the rules above) of its definition in Sect. 2, *i.e.* the function defined by

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

and the operator ? is a nondeterministic choice.

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

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

Their behaviors of "s", ? and $+\!\!+$ are the same as the reinterpretations of *text* "s", $\mathbin{<\!\!+}$ and $\mathbin{<\!\!>}$, respectively; we use different symbols to clarify that the conversion discards the pretty-printing semantics. Note that, since the language is linear and treeless, the call-time choice and the run-time choice [19] do not differ.

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). $\Gamma \vdash e \Downarrow_{\mathrm{ND}} s$ *iff* $\Gamma \vdash \underline{e} \Downarrow s$. □

Grammar-Based Inversion. The rewritten programs can be processed to obtain a grammar with actions[4] that computes the inverse of the rewritten program by using grammar-based inversion [15]. The basic idea of the inversion 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 original inputs

[4] In the original paper [15], transformations on parse trees (or more precisely, derivation trees of productions) are used, instead of semantic actions.

Rules of F_f

For function $\underline{f}$, we generate:

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

Rules of E_e

For expression $\underline{e}$, we generate:

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

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

$(t_1, \ldots, t_n)$. For an expression $\underline{e}$ such that $\Gamma \vdash \underline{e} \Downarrow s$, E_e is used to parse string s, and the semantic action returns the original environment Γ. The generation of the production rules and semantics actions are presented in Fig. 5. The grammar in Sect. 2 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)

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

Proof. Follows from [15]. □

Let *ppr* be a single-argument function defined in the core language, 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 $\underline{ppr}$ is injective. Note that the inversion can produce arbitrary CFGs, and therefore FliPpr requires parser generators that support *full* CFGs.

4 Surface Language: Making It More Flexible

The core language 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 that has a relaxed form of the restrictions, and through fusion techniques (specifically deforestation [21] or supercompilation [20]), 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.

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

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.

$$
\begin{array}{l}
\mathit{ppr}\ x = \mathit{ppr}_5\ x \qquad \texttt{-- 5 is the lowest precedence level} \\
\ldots \\
\mathit{ppr}_{-5}\ (\mathsf{Sub}\ x\ y) = \ldots \mathit{ppr}_6\ x \ldots \mathit{text}\ \texttt{"-"} \ldots \mathit{ppr}_7\ y \ldots \qquad \texttt{-- (1)} \\
\mathit{ppr}_{-5}\ (\mathsf{Div}\ x\ y) = \ldots \mathit{ppr}_5\ x \ldots \mathit{text}\ \texttt{"/"} \ldots \mathit{ppr}_6\ y \ldots \qquad \texttt{-- (2)} \\
\ldots \\
\mathit{ppr}_{-6}\ (\mathsf{Sub}\ x\ y) = \mathit{text}\ \texttt{"("} \diamond \mathit{nil} \diamond \ldots \texttt{\{-}\ \text{the RHS of (1)}\ \texttt{-\}} \ldots \diamond \mathit{nil} \diamond \mathit{text}\ \texttt{")"} \\
\mathit{ppr}_{-6}\ (\mathsf{Div}\ x\ y) = \ldots \texttt{\{-}\ \text{the RHS of (2)}\ \texttt{-\}} \ldots \\
\ldots \\
\mathit{ppr}_{-7}\ (\mathsf{Sub}\ x\ y) = \mathit{text}\ \texttt{"("} \diamond \mathit{nil} \diamond \ldots \texttt{\{-}\ \text{the RHS of (1)}\ \texttt{-\}} \ldots \diamond \mathit{nil} \diamond \mathit{text}\ \texttt{")"} \\
\mathit{ppr}_{-7}\ (\mathsf{Div}\ x\ y) = \mathit{text}\ \texttt{"("} \diamond \mathit{nil} \diamond \ldots \texttt{\{-}\ \text{the RHS of (2)}\ \texttt{-\}} \ldots \diamond \mathit{nil} \diamond \mathit{text}\ \texttt{")"}
\end{array}
$$

There are a lot of undesirable repetitions in the above definition largely due to the treeless restriction.

Another problem that it is non-trivial to separate variable names with predefined names. For example, let us consider pretty-printing for $\mathsf{Var}\ x$. One may be tempted to write $\mathit{ppr\_}\ (\mathsf{Var}\ x) = \mathit{text}\ x$ 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.

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

Note that strings are represented as lists of characters as in Haskell. This function *ppr_* is partial and intentionally undefined for `Var "-"`. In this definition, we have successfully restricted variable names to range over lower-case English alphabets, but in a very cumbersome way.

4.2 An Overview

To reduce the programming effort, we propose a surface language, 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.

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

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

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

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, which enable users to avoid duplicating frequently-occurring patterns such as $\mathit{text}\ \texttt{"("} \mathbin{<\!\!>} \mathit{nil} \mathbin{<\!\!>} \ldots \mathbin{<\!\!>} \mathit{nil} \mathbin{<\!\!>} \mathit{text}\ \texttt{")"}$.
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 [14,20,21] 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.

4.3 Surface Language

Figure 6 shows the syntax of the surface language. 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 operations 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 partial-evaluation, and variable uses are linear. Formally, primitive types τ and function types σ are defined by:

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

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

Fig. 6. Syntax of the surface language: *pred* are Boolean predicates

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

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

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

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

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

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

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

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

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

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

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

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 7 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 [1]. A program is assumed to have a distinguished entry point function of type AST → 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 to a typed one: only arguments of type AST or Doc are restricted to be variables. Moreover, we view programs in the surface language as *multi-tier* systems [14]: 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 topologically 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. 4.2 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 the surface language, as it is a straightforward extension of the core language. Similar to the case of the core language, 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 the Core Language

The surface language is elaborated to the core language 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.

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. 4.2, 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.

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, a standard partial evaluation suffices to eliminate all the St-expressions and thus we omit the

details. For the example in Sect. 4.2, we obtain the partially evaluated functions as below.

$$
\begin{aligned}
&ppr\ x = go_5\ x\\
&\ldots\\
&go_5\ (\mathsf{Sub}\ x\ y) = \ldots go_5\ x \ldots go_6\ y \ldots\\
&go_5\ (\mathsf{Div}\ x\ y) = \ldots go_6\ x \ldots go_7\ y \ldots\\
&\ldots\\
&go_6\ (\mathsf{Sub}\ x\ y) = \ldots go_5\ x \ldots go_6\ y \ldots\\
&go_6\ (\mathsf{Div}\ x\ y) = par\ (\ldots go_6\ x \ldots go_7\ y \ldots)\\
&\ldots\\
&go_7\ (\mathsf{Sub}\ x\ y) = par\ (\ldots go_5\ x \ldots go_6\ y \ldots)\\
&go_7\ (\mathsf{Div}\ x\ y) = par\ (\ldots go_6\ x \ldots go_7\ y \ldots)
\end{aligned}
$$

Roughly speaking, thanks to the type $\mathsf{AST} \to \mathsf{Doc}$ 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.

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 [21]. Roughly speaking, deforestation (or, supercompilation [20][5]) 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 (<+>).
- $\mathcal{D}[\![f\ \widetilde{x}\ \widetilde{e}]\!] = f_{\widetilde{e}}\ \widetilde{x}\ \widetilde{z}$. 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 [21], 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. 4.2.

[5] Because of the linearity, Wadler's deforestation [21] and (positive) supercompilation [20] coincide for the surface language.

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

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

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

$$go_5\ x = \mathit{manyPars}\ (go_{\_5}\ x)$$

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

$$\begin{aligned} &go_5\ x &&= \mathit{manyPars}_{go_{\_5}\ x}\ x \\ &\mathit{manyPars}_{go_{\_5}\ x}\ x &&= go_{\_5}\ x \mathbin{<\!\!+} \mathit{parMP}_{go_{\_5}\ x}\ x \\ &\mathit{parMP}_{go_{\_5}\ x}\ x &&= \mathsf{text}\ \texttt{"("} \mathbin{<\!\!>} \mathit{nil} \mathbin{<\!\!>} go_{\_5}\ x \mathbin{<\!\!>} \mathit{nil} \mathbin{<\!\!>} \mathsf{text}\ \texttt{")"} \end{aligned}$$

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

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 [18]. Note again that call-by-value and call-by-name coincide in our surface language.

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 2 can be generalized and $\mathcal{D}[\![e]\!]$ terminates for tier-n expression e. Also, since deforestation (supercompilation) is a sort of partial-evaluation, the steps (2) and (3) of the transformation can be performed at once. We omit a formal discussion on this for space reason.

5 An Involved 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 the core language in the surface language, which, if fed to FliPpr, 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* `"s"` 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 = pprRules x

pprRules []       = nil
pprRules (r : rs) = nil <> pRules r rs <> nil

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

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

```
pprRule (Rule f ps e) =
  group (var f <> space <> pprPats ps <> space' <> text "=" <> nest 4 (line' <> pprExp e))
var x = text (x as [a-z][_a-zA-Z0-9]*'*)
```

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

```
pprPats []       = text ""
pprPats (p : ps) = pPats p ps

pPats p' []       = pprPat p'
pPats p' (p : ps) = pprPat p' <> space <> pPats p ps
```

Redundant parentheses in patterns are admissible to 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 <> space <> 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 <> space <> pExps e es)
go_ i (EOp OAlt e1 e2) =
    parIf (i ≥ 5) (group (go 5 e1 <> nest 2 (line' <> text "<+" <> space' <> go 4 e2)))
go_ i (EOp OCat e1 e2) =
    parIf (i ≥ 6) (group (go 6 e1 <> nest 2 (line' <> text "<>" <> space' <> go 5 e2)))
go_ i (EVar f [])       = var f
go_ i (EVar f (e : es)) = parIf (i ≥ 9) (var f <> space <> pExps e es)
```

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

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

6 Discussion

We discuss limitations and extensions of FliPpr.

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* $(f\ x\ \mathbf{as}\ r)$ 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 and parsing.

Higher-Order Functions. Higher-order functions, such as `map`, `foldr` and `foldr1` are useful in writing pretty-printers. For example, *pprRules* and *pprPats* in Sect. 5 can be more conveniently implemented by `map` and `foldr1`. However, general use of higher-order functions in pretty-printing may produce grammars that go beyond CFG. The linearity restriction is also affected, most of the higher-order functions use the functional arguments more than once on the right-hand sides.

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. In other words, functions must be *treeless* in the sense that intermediate function values are prohibited, and all the higher-order values must be variables (function names). We leave this extension as future work.

Spacing. We have demonstrated that careful use of whitespaces in the definition of the pretty-printer is an effectively way to control the behavior of the generated parser. For example, for pretty-printing constructor application in Sect. 5, we wrote (*con c* <> *space* <> *pExps e es*); the use of *space* (representing 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. But it may require some major surgery to the current system.

Non-Linearity. In the literature of tree transducers [9], the discussion of linearity can be separated into input- and output-linearity. In our case, variables of type AST can be seen as inputs, and those of type Doc can be seen as outputs.

For AST 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 [13] (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. In the absence of 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 [15], but call-by-name is suitable for deforestation [18]. 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.

7 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 [2]. 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 [17] based on invertible programming, and BNFC-meta [4] based on meta programming. Both work recognizes the importance of good printing, but is not able to support pretty-printing.

There are also general-purpose bidirectional languages [3, 6, 11] that in theory can be used to build the printer/parser pair from the definition of one of them. Notably quotient lenses [7] 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 practically execution. In particular, the pretty-printing libraries of Wadler [22] and Hughes [10] 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 [8, 16]. 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 [15], which generates grammars and outsources the parsing algorithms to selected parser generators.

There are a lot of discussions on how to make deforestation (supercompilation) terminate (*e.g.*, [12]) 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 the nested calls are fused away, has not been the focus of study in

the literature. Notable exceptions are Wadler's original work [21] and tree transducer fusion [1, 9, 14]. 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 directed applied in our case.

8 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 [22], 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 [15] to produce a CFG parser. For the inversion to be possible, the language is restricted to be linear and treeless [21]. 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 the example of runlength encoding, we want to decode both `A3B1` and `A1A2B1` as `AAAB`, but an encoder "prefers" the former. Our result for pretty-printing/parsing benefits from Wadler's combinators, in which the "preference" is encapsulated in the combinators in a compositional way. It is an interesting problem to see how the technique may apply in different contexts.

Acknowledgments. 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 Number 24700020. 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.

References

1. Baker, B.S.: Composition of Top-down and Bottom-up Tree Transductions. Information and Control 41(2), 186–213 (1979)
2. Boulton, R.J.: Syn: A Single Language for Specifiying Abstract Syntax Tress, Lexical Analysis, Parsing and Pretty-Printing. Technical Report UCAM-CL-TR-390, University of Cambridge Computer Laboratory (1996)
3. Brabrand, C., Møller, A., Schwartzbach, M.I.: Dual Syntax for XML Languages. Inf. Syst. 33(4-5), 385–406 (2008)
4. Duregård, J., Jansson, P.: Embedded Parser Generators. In: Haskell 2011: Proceedings of the 2011 ACM SIGPLAN Haskell Symposium, pp. 107–117. ACM (2011)

5. Duregård, J., Jansson, P., Wang, M.: Feat: Functional Enumeration of Algebraic Types. In: Haskell 2012: Proceedings of the 2012 ACM SIGPLAN Haskell Symposium, pp. 61–72. ACM (2012)
6. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
7. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient Lenses. In: ICFP 2008: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 383–396. ACM (2008)
8. Frost, R.A., Hafiz, R., Callaghan, P.: Parser Combinators for Ambiguous Left-Recursive Grammars. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 167–181. Springer, Heidelberg (2008)
9. Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers, 1st edn. Springer-Verlag New York, Inc., Secaucus (1998)
10. Hughes, J.: The Design of a Pretty-Printing Library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)
11. Jansson, P., Jeuring, J.: Polytypic Data Conversion Programs. Sci. Comput. Program. 43(1), 35–75 (2002)
12. Jonsson, P.A., Nordlander, J.: Positive Supercompilation for a Higher Order Call-by-Value Language. In: POPL 2009: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 277–288. ACM (2009)
13. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-Order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification. In: POPL 2010: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 495–508. ACM (2010)
14. Kühnemann, A., Glück, R., Kakehi, K.: Relating Accumulative and Non-accumulative Functional Programs. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 154–168. Springer, Heidelberg (2001)
15. Matsuda, K., Mu, S.-C., Hu, Z., Takeichi, M.: A Grammar-Based Approach to Invertible Programs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 448–467. Springer, Heidelberg (2010)
16. Might, M., Darais, D., Spiewak, D.: Parsing with Derivatives: A Functional Pearl. In: ICFP 2011: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, pp. 189–195. ACM (2011)
17. Rendel, T., Ostermann, K.: Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In: Haskell 2010: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium, pp. 1–12. ACM (2010)
18. Sands, D.: Proving the Correctness of Recursion-Based Automatic Program Transformations. Theor. Comput. Sci. 167(1&2), 193–233 (1996)
19. Søndergaard, H., Sestoft, P.: Non-Determinism in Functional Languages. Comput. J. 35(5), 514–523 (1992)
20. Sørensen, M.H., Glück, R., Jones, N.D.: A Positive Supercompiler. J. Funct. Program. 6(6), 811–838 (1996)
21. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
22. Wadler, P.: A Prettier Printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming. Palgrave Macmillan (2003)