Embedding Invertible Languages with Binders

A Case of the FliPpr Language

Kazutaka Matsuda
Tohoku University
Japan
kztk@ecei.tohoku.ac.jp

Meng Wang
University of Bristol
United Kingdom
meng.wang@bristol.ac.uk

Abstract

This paper describes a new embedding technique of invertible programming languages, through the case of the FliPpr language. Embedded languages have the advantage of inheriting host languages' features and supports; and one of the influential methods of embedding is the tagless-final style, which enables a high level of programmability and extensibility. However, it is not straightforward to apply the method to the family of invertible/reversible/bidirectional languages, due to the different ways functions in such domains are represented. We consider FliPpr, an invertible pretty-printing system, as a representative of such languages, and show that Atkey et al.'s unembedding technique can be used to address the problem. Together with a reformulation of FliPpr, our embedding achieves a high level of interoperability with the host language Haskell, which is not found in any other invertible languages. We implement the idea and demonstrate the benefits of the approach with examples.

CCS Concepts • **Software and its engineering → Functional languages**; **Domain specific languages**; *Polymorphism*; *Syntax*; *Parsers*;

Keywords EDSL, Program Inversion, Pretty-Printing, Parsing

ACM Reference Format:
Kazutaka Matsuda and Meng Wang. 2018. Embedding Invertible Languages with Binders: A Case of the FliPpr Language. In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27-28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3242744.3242758

Haskell '18, September 27-28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242758

1 Introduction

Embedded languages, languages expressed via libraries in host languages, are popular. The great advantage of the approach is that an embedded language inherits the generic features of its host language, as well as the ecosystem including compilers, editors, IDEs etc. Haskell, featuring strong abstraction mechanisms (higher-order functions, type classes and so on) and a powerful type system, serves as a good platform for embedded languages [6, 8, 16, 18, 20, 25, 33, 36].

To embed a language, one needs to specify a way to express the guest language's constructs in the host language. Among the various ways to embed syntax, the *tagless-final* style [4] is known for its programmability and extensibility. For example, for the simply-typed λ-calculus, one can express its syntax by the following type class.

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Lam}\ e\ \textbf{where} \\
\quad abs :: (e\ a \to e\ b) \to e\ (a \to b) \\
\quad app :: e\ (a \to b) \to e\ a \to e\ b
\end{array}
$$

In the tagless-final style, guest-language binders are expressed by host-language functions, enabling the construction of embedded terms via the host language's (higher-order) functions. In this case, the functions *abs* and *app* provide a way to interconvert functions of the two levels. The semantics of the guest language are given by instances of the type class. For example, using the identity functor is enough for the usual evaluation (where the guest and host language functions coincide).

One may take this promotion of the guest-language's binders to host-language functions for granted, and indeed for most cases straightforward definitions of *abs* and *app* exist. However for some semantic domains, where one language's functions are not naturally the other's, such promotion becomes much more difficult to implement. One typical class of examples are the invertible/reversible/bidirectional programming languages [10, 24, 29, 37], where a "function" can be executed in both forward and backward directions (in the remainder of this paper, we will use "invertible languages" to refer to this class of languages). In this case, a function in such a domain is actually a (encapsulated) pair of functions (one in each direction) in a conventional unidirectional language, which is problematic as a bound value serves as an input of one function and output of the other, but the tagless-final style expects the binder to be realized by a single function.

This mismatch of function representations creates a barrier in effective embedding. To the best of the authors knowledge, existing embedded implementations of invertible programming languages, such as implementations (e.g., `lens`, `pointless-lenses` [30] and `putlenses` [31]) of lens [10] variants and `invertible-syntax` [34], do not have any binders. As a result, the ability to construct invertible programs using host language functions is limited: programming with names and binders is simply unavailable and strictly point-free composition is the only means for program construction. In addition to the problem of binder promotion, invertible languages sometimes treat recursive definitions explicitly for efficiency or safety [10, 12, 13, 23, 24, 29], and there are very often specialized syntactic restrictions [23, 24, 29]. These characteristics together make the embedding of inverting languages particularly challenging.

In this paper, we provide a solution to the problem exemplified by embedding in Haskell an invertible programming language FliPpr [24]. FliPpr is a language for writting pretty-printers based on Wadler [36]'s pretty-printing combinators, which can be inverted to produce parsers that are correct with respect to the pretty-printers:

$$\textit{parse}\ (\textit{prettyprint}\ t) = t \qquad \text{(Correctness Law)}$$

We choose FliPpr as it is representative of an invertible language and is independently useful. It focuses on a certain application (development of pretty-printers and parsers in synchronization), and adopts syntactic restrictions to enable (outsourced) complex analysis. Specifically, the FliPpr system generates context-free grammar with actions that is compatible with existing parsing algorithms and tools. Moreover, FliPpr features a conventional programming style, allowing function definitions, calls and pattern matching; the preservation of them in the embedding is a challenge that is of general interest.

The key idea underpinning our approach is the use of a technique known as *unembedding* [1, 2, 19], which transforms syntax in a tagless-final style to de Bruijn terms. The original motivation of unembedding is to convert a user-friendly syntax to a program-manipulation-friendly form. We show in this paper that the same technique is useful for embedding invertible languages as it gives access to type environments via de Bruijn terms. Of course, to embed an invertible language requires more than unembedding. In addition, we reformulate FliPpr to accept arbitrary user-defined Haskell datatypes, and deal with features such as syntactic restrictions and explicit handling of recursions.

As far as we are aware, this work is the first dedicated effort to embed an invertible language with enhanced interoperability, and we have successfully embedded FliPpr to achieve a good result. Despite the success, we also recognize that there are significant differences among different invertible languages, and the techniques proposed in this paper alone will not be sufficient for embedding arbitrary invertible languages. Nevertheless, we believe that the progress made in this paper is a significant step towards a general solution.

In summary, our contributions are:

- We use the unembedding transformation [1, 2, 19] to address the binder representation problem in embedding invertible programming languages.
- We reformulate FliPpr to enhance its interoperability with Haskell. Specifically, FliPpr functions now can take arbitrary Haskell datatypes as input.
- We discuss how we treat rather complex features in FliPpr, including the treeless restriction [35] and explicit treatment of recursions to produce CFGs with actions.

The implementation of the system is available from https://bitbucket.org/kztk/flippre/.

The rest of this paper is organized as follows. Section 2 reviews the techniques our paper relies on, namely FliPpr and the unembedding transformation. Section 3 describes how we embed non-recursive FliPpr, including its reformulation. Section 4 shows our treatment of recursions. Section 5 proposes improvements of our embedding both from programmability and efficiency aspects. Section 7 explores embedding of general invertible languages and discusses related work, and Section 8 concludes the paper.

2 Preliminaries

In this section, we briefly review the techiniques on which our paper is based on, namely FliPpr [24] and the unembedding transformation [1, 2, 19].

2.1 FliPpr: Invertible Pretty-Printing System

FliPpr is a system that derives a parser from a pretty-printer definition so that the parser is correct with respect to the printer. The FliPpr system has a designated programming language, which is also called FliPpr, that is based on Wadler's pretty-printing combinators [36]. FliPpr guarantees that it always generates a parser representable as a context-free grammar (CFG) with actions.

More specifically, FliPpr consists of two languages: a core and a surface one. The core language has strong syntactic restrictions, namely being *treeless* and *linear*, and is directly subjected to inversion [23]. The surface language is translated to the core via program transformations such as deforestation [35].

For example, let us consider a subset of arithmetic expressions that consist of only "1" and "−". A pretty-printer in Fig. 1 is written in the surface language, which will be converted to the pretty-printer in Fig. 2. After that, the FliPpr system generates the CFG in Fig. 3.

```
pprMain x = nil <> ppr False x <> nil
ppr b x = manyParens (ppr' b x)
ppr' b One       = text "1"
ppr' b (Sub x y) = parensIf b (group (
   ppr False x <>
   nest 2 (lineN <> text "-" <> spaceN <> ppr True y)))
manyParens d = d <? parens (manyParens d)
parens d = text "(" <> nil <> d <> nil <> text ")"
parensIf b d = if b then parens d else d
```

Figure 1. A Program in FliPpr’s Surface Language.

```
pprMain x = nil <> ppr_F x <> nil
ppr_F x = ppr'_F x <? text "(" <> nil <> ppr_F x <> nil <> text ")"
ppr'_F One       = text "1"
ppr'_F (Sub x y) = group (
   ppr_F x <> nest 2 (lineN <> text "-" <> spaceN <> ppr_T y))
ppr_T x = ppr'_T x <? text "(" <> nil <> ppr_T x <> nil <> text ")"
ppr'_T One       = text "1"
ppr'_T (Sub x y) = text "(" <> nil <> group (
   ppr_F x <>
   nest 2 (lineN <> text "-" <> spaceN <> ppr_T y)) <> nil <> text ")"
```

Figure 2. A Corresponding Program in the Core Language.

```
PprMain → Nil Ppr_F Nil    {$2}
Ppr_F → Ppr'_F             {$1}
      → "(" Nil Ppr_F Nil ")"   {$3}
Ppr'_F → "1"                    {One}
       → Ppr_F LineN "-" SpaceN Ppr_T   {Sub $1 $5}
...
```

Figure 3. The CFG with actions obtained from Fig. 2

$$
\begin{array}{lll}
prog & ::= & r_1 \ldots r_n \\
r & ::= & f\ p_1\ \ldots\ p_n = e \\
e & ::= & \mathit{text}\ s \mid e_1 \mathbin{<\!>} e_2 \mid \mathit{line} \mid \mathit{nest}\ n\ e \mid \mathit{group}\ e \\
 & \mid & e_1 \mathbin{<?} e_2 \mid f\ x_1\ \ldots\ x_n \\
p & ::= & x \mid \mathsf{C}\ p_1 \ldots p_n
\end{array}
$$

Figure 4. The syntax of the core language of FliPpr

In this paper, we only focus on the core language and its parser generation semantics because (1) the host language, Haskell, will provide the functionalities of the surface language and thus eliminate the need of it, and (2) the part about pretty-printing semantics is rather straightforward, involving only Wadler’s combinators, and is thus omitted.

2.1.1 The Core Language of FliPpr

Figure 4 gives the syntax of the core language. A program consists of rules of the form $f\ p_1\ \ldots\ p_n = e$, where each p_i is a pattern defined in the standard way and e is an expression. An expression ranges over Wadler’s combinators ($\mathit{text}\ s$, $e_1 <\!> e_2$, line, $\mathit{nest}\ n\ e$ and $\mathit{group}\ e$), biased choice $e_1 <?\ e_2$, and treeless [35] function call $f\ x_1\ \ldots\ x_n$. Here, being treeless means that only variables can serve as arguments of functions. The programs are expected to be linear, which is checked statically in the original FliPpr, but at run-time in this paper.

We briefly explain the behavior of Wadler’s combinators [36] in pretty-printing.

- $\mathit{text}\ s$ renders the string s in pretty-printing.
- $e_1 <\!> e_2$ represents concatenation.
- line represents a new line with indentation according to the current indentation level. However, when placed under group, it can also be rendered as a single space.
- $\mathit{nest}\ n\ e$ increases the current indentation level by n in e.
- $\mathit{group}\ e$ smartly chooses layouts in e with line that either works as a new line or a single space.

More specifically, $\mathit{group}\ e$ says that “render e as horizontal as possible, and otherwise render e with newlines with appropriate indentation indicated by nest”. Thus, the pretty-printer in Fig. 1 and 2 will print Sub (Sub One One) (Sub One One) as

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

depending on the screen width, assuming that nil, lineN and spaceN behaves as text "", line and text " ", respectively.

Parsing semantics is designed to parse all the printer’s outputs, and in addition non-pretty strings which cannot be produced by a pretty-printer but are nevertheless valid grammatically. To achieve this, FliPpr reinterprets Wadler’s combinators in the parsing direction to accommodate a variety of strings. Specifically, line is reinterpreted as arbitrary non-empty spaces, and nest and group are simply ignored because they only affect the behavior of lines. Moreover, a new operator $e_1 <?\ e_2$ is introduced to admit redundant spaces and extra parentheses in places. Intuitively, $e_1 <?\ e_2$ means that e_2 is also a valid string representation but e_1 is prettier. That is, in pretty-printing, the operator simply ignores e_2 and behaves as e_1, but in parsing, it behaves as nonderministic choice between e_1 and e_2. The derived operators nil, lineN, and spaceN we have seen in Figure 1 are defined by using <? as below.

```
white = text " " <? text "\n"  -- parses a white space
space = white <> nil           -- parses one-or-more whites
nil = text "" <? space         -- parses zero-or-more whites
lineN = line <? text ""        -- parses zero-or-more whites
spaceN = space <? text ""      -- parses zero-or-more whites
```

Notice that the last three functions have the same behavior in parsing, but not in pretty-printing.

As a result, non-pretty strings such as "␣␣(1␣␣-(1␣␣))" become parsable, together with the pretty ones we have seen above.

2.1.2 Parser-Generation Semantics

A CFG with actions is generated from a pretty-printer definition written in the FliPpr language. From a program in the FliPpr core language, the following steps are taken for the generation.

1. Prepare nonterminals F_f and E_e for each function f and expression e in a program, where parsing results of F_f and E_e's are arguments of f and a value environment for e such that they evaluates to a parsed string, respectively.
2. For each rule $f\ p_1\ \dots\ p_n = e$, add the rule:
$$F_f \to E_e \quad \{\mathbf{let}\ \theta = \$1\ \mathbf{in}\ (p_1\theta, \dots, p_n\theta)\}.$$
3. For each expression e, add the rule(s) as below.
   - When $e = \mathit{text}\ s$, add:
   $$E_{\mathit{text}\ s} \to s \quad \{\emptyset\}$$
   - When $e = e_1 <> e_2$, add:
   $$E_{e_1 <> e_2} \to E_{e_1}\ E_{e_2} \quad \{\$1 \cup \$2\}$$
   - When $e = \mathit{line}$, add:
   $$E_{\mathit{line}} \to \mathit{White}^+ \quad \{\emptyset\}$$
   - When $e = \mathit{nest}\ n\ e'$ or $e = \mathit{group}\ e'$, add:
   $$E_e \to E_{e'} \quad \{\$1\}$$
   - When $e = e_1 <? e_2$, add:
   $$E_{e_1 <? e_2} \to E_{e_1} \quad \{\$1\}$$
   $$E_{e_1 <? e_2} \to E_{e_2} \quad \{\$1\}$$
   - When $e = f\ x_1\ \dots\ x_n$, add:
   $$E_{f\ x_1\ \dots\ x_n} \to F_f \quad \left\{\begin{array}{l}\mathbf{let}\ (v_1, \dots, v_n) = \$1 \\ \mathbf{in}\ \{x_1 = v_1, \dots, x_n = v_n\}\end{array}\right\}$$

Here, White^+ is the nonterminal that generates non-empty white spaces.

The parser is correct with respect to the (Correctness Law) [24].

2.2 Unembedding Transformation

In this section, we review the unembedding transformation [2, 19], which transforms syntax in a tagless-final style [4] to de Bruijn terms. We use the simply-typed λ calculus as an example, whose syntax in the tagless-final style is already given in Section 1. The de Bruijn terms are represented by the following GADTs.[1]

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{DLam}\ \Gamma\ a\ \mathbf{where} \\
\quad \mathsf{Var} :: \mathsf{In}\ a\ \Gamma \to \mathsf{DLam}\ \Gamma\ a \\
\quad \mathsf{Abs} :: \mathsf{DLam}\ (\Gamma, a)\ b \to \mathsf{DLam}\ \Gamma\ (a \to b) \\
\quad \mathsf{App} :: \mathsf{DLam}\ \Gamma\ (a \to b) \to \mathsf{DLam}\ \Gamma\ a \to \mathsf{DLam}\ \Gamma\ b \\
\mathbf{data}\ \mathsf{In}\ a\ \Gamma\ \mathbf{where} \\
\quad \mathsf{Z} :: \mathsf{In}\ a\ (\Gamma, a) \\
\quad \mathsf{S} :: \mathsf{In}\ a\ \Gamma \to \mathsf{In}\ a\ (\Gamma, b)
\end{array}
$$

Converting de Bruijn terms to the tagless-final style is rather easy. However, the opposite is not straightforward because we need to recover the type environment information.

The conversion is realized by preparing an instance of Lam; to do so, we prepare the following datatypes.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{U}\ a = \mathsf{U}\ \{\mathit{unU} :: \forall\Gamma.\mathsf{TEnv}\ \Gamma \to \mathsf{DLam}\ \Gamma\ a\} \\
\mathbf{data}\ \mathsf{TEnv}\ \Gamma\ \mathbf{where} \\
\quad \mathsf{TEmp} :: \mathsf{TEnv}\ () \\
\quad \mathsf{TExt} :: \mathsf{TEnv}\ \Gamma \to \mathsf{Proxy}\ a \to \mathsf{TEnv}\ (\Gamma, a)
\end{array}
$$

The type U a essentially represents the dependent product $\prod_\Gamma.\mathsf{DLam}\ \Gamma\ a$, but since Haskell does not allow value-level pattern matching on types, we pass Γ's value-level representation TEnv Γ instead. The datatype Proxy is a phantom type defined as **data** Proxy a = Proxy in `Data.Typeable`.

Then, we are ready to give an instance of Lam. It is rather straightforward to define *app*, which just passes TEnv Γ.

$$
\begin{array}{l}
\mathbf{instance}\ \mathsf{Lam}\ \mathsf{U}\ \mathbf{where} \\
\quad \mathit{app}\ (\mathsf{U}\ f)\ (\mathsf{U}\ a) = \mathsf{U}\ (\lambda\gamma \to \mathsf{App}\ (f\ \gamma)\ (a\ \gamma))
\end{array}
$$

However, the definition of *abs* is much trickier. Its basic structure is as below.

$$
\begin{array}{l}
\mathit{abs}\ f = \mathsf{U}\ \$\ \lambda\gamma \to \\
\quad \mathbf{let}\ \gamma_a = \mathsf{TExt}\ \gamma\ \mathsf{Proxy} \\
\quad \mathbf{in}\ \mathsf{Abs}\ (\mathit{unU}\ (f\ (\mathsf{U}\ \$\ \lambda\gamma' \to \mathsf{Var}\ ???)))\ \gamma_a
\end{array}
$$

Since the Abs's argument must have the type DLam $(\Gamma, a)\ b$, we pass γ_a :: TEnv (Γ, a) to the result of f :: U $a \to$ U b to obtain a value of the type. But, the problem comes in the ??? part, which must have type In $a\ \Gamma'$ where Γ' comes from the argument γ' :: TEnv Γ'. We must convert the variable Z :: In $a\ (\Gamma, a)$ introduced by Abs to In $a\ \Gamma'$.

Atkey [1] proved by parametricity that Γ' must be as big as (Γ, a); that is, $\Gamma' = ((\dots((\Gamma, a), a_1), \dots), a_n)$ for some n. Intuitively, this says that f must be a context consisting of Abs, App, Var and a hole, and thus its argument can only occur in a deeper position. This suggests a coercion *coer* :: TEnv $\Gamma \to$ TEnv $\Gamma' \to$ In $a\ \Gamma \to$ In $a\ \Gamma'$ that applies S n times to complete the definition.

$$
\begin{array}{l}
\mathit{abs}\ f = \mathsf{U}\ \$\ \lambda\gamma \to \\
\quad \mathbf{let}\ \gamma_a = \mathsf{TExt}\ \gamma\ \mathsf{Proxy} \\
\quad \mathbf{in}\ \mathsf{Abs}\ (\mathit{unU}\ (f\ (\mathsf{U}\ \$\ \lambda\gamma' \to \mathsf{Var}\ (\mathit{coer}\ \gamma\ \gamma'\ \mathsf{Z}))))\ \gamma_a
\end{array}
$$

The coercion function fails if Γ' is not as big as (Γ, a), but such a case cannot happen due to parametricity [1]. Thus, the following conversion is indeed *total*.

$$
\begin{array}{l}
\mathit{unemb} :: (\forall e.\mathsf{Lam}\ e \Rightarrow e\ a) \to \mathsf{DLam}\ ()\ a \\
\mathit{unemb}\ (\mathsf{U}\ e) = e\ \mathsf{TEmp}
\end{array}
$$

[1] The code is actually incorrect in Haskell, as Γ is an uppercase letter and cannot be recognized as a (type) variable. We abuse the notation to emphasize that Γ represents a typing environment.

We omit the definition of *coer* (see Appendix A.1).

3 Embedding Non-Recursive FliPpr

This and the next sections discuss embedding of FliPpr by using the unembedding transformation [2]. This section focuses on non-recursive programs and a slight reformulation of FliPpr so that it can pretty-print arbitrary user-defined Haskell datatypes. The treatment of recursion is left for Section 4.

3.1 Interoperable FliPpr

We first reformulate FliPpr so that it admits user-defined Haskell types. We replace global function definitions with λ abstractions/applications while keeping the treeless restriction. Also, we give a semantics based on parser combinators.

3.1.1 New Syntax and Type System

The new syntax of FliPpr is as below.

$$\begin{array}{rl} e ::= & \lambda x.e \mid e\ x \mid \mathit{text}\ s \mid e_1 \mathbin{\texttt{<>}} e_2 \mid e_1 \mathbin{\texttt{<?}} e_2 \\ \mid & \textbf{case}\ x\ \textbf{of}\ \{(\phi_i \to x_i) \to e_i\}_i \\ \mid & \textbf{let}\ () = x\ \textbf{in}\ e \mid \textbf{let}\ (x_1, x_2) = x\ \textbf{in}\ e \end{array}$$

For simplicity, we omit *line*, *nest* n e and *group* e because their treatments are straightforward (from a parsing perspective). Here, pattern-matching functionality in the original core language is separated into case analysis by **case** and decomposition by **let**, where ϕ_i in **case** is a (expected to be decidable) partial injection of which failure indicates that the pattern $(\phi_i \to x_i)$ does not match. Notice that the language still has the treeless restriction; the second operand of a function application must be a variable, and scruitinee expressions must also be variables.

The language has the following types.

$$\begin{array}{lll} \tau ::= \mathsf{D} \mid \iota \to \tau & & \text{(first-order printer types)} \\ \iota ::= \text{(Haskell's datatypes)} & & \text{(input types)} \end{array}$$

Notice that ι can be any Haskell datatype as it will be manipulated by ϕ that also comes from Haskell. The typing rules are shown in Fig. 5. The judgment $\Gamma \vdash e : \tau$ reads that, under type environment Γ, e has type τ, where Γ maps variables to ι types. Here, PartialInj is defined by

$$\textbf{type}\ \mathsf{PartialInj}\ \iota\ \iota' = (\iota \to \mathsf{Maybe}\ \iota', \iota' \to \iota)$$

representing partial injections.

3.1.2 Semantics of New FliPpr

We give its semantics based on Haskell programs with applicative [28] parser combinators. We assume a parser type Parser a and the following combinators.

- $(\langle\$\rangle) :: (a \to b) \to \mathsf{Parser}\ a \to \mathsf{Parser}\ b$
- $(\langle*\rangle) :: \mathsf{Parser}\ (a \to b) \to \mathsf{Parser}\ a \to \mathsf{Parser}\ b$
- $\mathit{ptext} :: \mathsf{String} \to \mathsf{Parser\ String}$
- $\mathit{pfail} :: \mathsf{Parser}\ a$
- $(\langle|\rangle) :: \mathsf{Parser}\ a \to \mathsf{Parser}\ a \to \mathsf{Parser}\ a$

$$\frac{\Gamma, x : \iota \vdash e : \tau}{\Gamma \vdash \lambda x.e : \iota \to \tau} \qquad \frac{\Gamma \vdash e : \iota \to \tau \quad \Gamma(x) = \iota}{\Gamma \vdash e\ x : \tau}$$

$$\frac{\{\Gamma \vdash e_i : \mathsf{D}\}_i \quad op \in \{\mathit{text}\ s, (\texttt{<>}), (\texttt{<?})\}}{\Gamma \vdash op\ e_1\ \ldots\ e_n : \mathsf{D}}$$

$$\frac{\Gamma(x) = \iota \quad \{\phi_i :: \mathsf{PartialInj}\ \iota\ \iota' \quad \Gamma, x_i : \iota' \vdash e_i : \mathsf{D}\}_i}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \{(\phi_i \to x_i) \to e_i\}_i : \mathsf{D}}$$

$$\frac{\Gamma(x) = () \quad \Gamma \vdash e : \tau}{\Gamma \vdash \textbf{let}\ () = x\ \textbf{in}\ e : \tau} \qquad \frac{\Gamma(x) = (\iota_1, \iota_2) \quad \Gamma, x_1 : \iota_1, x_2 : \iota_2 \vdash e : \tau}{\Gamma \vdash \textbf{let}\ (x_1, x_2) = x\ \textbf{in}\ e : \tau}$$

Figure 5. Typing Rules

Here, $\langle\$\rangle$ is *fmap*; $p_1 \langle*\rangle p_2$ parses the concatenation of p_1 and p_2 and then applies a parsing result of p_1 to that of p_2; *ptext* s parses s and returns s itself; *pfail* always fails; and $p_1 \langle|\rangle p_2$ nondeterministically choose between p_1 and p_2. We do not assume any concrete implementation of these combinators, but state that Parser a with the combinators denotes non-recursive CFGs.

Then, we look at the translation of terms-in-context.

$$[\![\Gamma \vdash e : \tau]\!] :: \mathsf{Sem}_{\Gamma,\tau}$$

We are expecting the following two isomorphisms on Sem:

$$\mathsf{Sem}_{\Gamma,\mathsf{D}} \sim \mathsf{Parser}\ [\![\Gamma]\!] \qquad \mathsf{Sem}_{\Gamma,\iota\to\tau} \sim \mathsf{Sem}_{\Gamma,x:\tau,\tau}$$

The former isomorphism says that a D-typed expression will be translated to a parser of which the parsing results are the values of the free variables in it. The latter says that Sem must have a “closed” structure to have abstractions and applications. Following this observation we can define $\mathsf{Sem}_{\Gamma,\tau} = \mathsf{Parser}\ (\mathsf{R}\ [\![\Gamma]\!]\ \tau)$. Here, $[\![\Gamma]\!]$ is defined as:

$$[\![\emptyset]\!] = () \qquad [\![\Gamma, x : \iota]\!] = ([\![\Gamma]\!], \mathsf{Maybe}\ \iota)$$

Accordingly, R, representing parsing results, is defined as:

$$\begin{array}{l} \textbf{data}\ \mathsf{R}\ a\ \tau\ \textbf{where} \\ \quad \mathsf{ResD} :: a \to \mathsf{R}\ a\ \mathsf{D} \\ \quad \mathsf{ResF} :: \mathsf{Eq}\ \iota \Rightarrow \mathsf{R}\ (a, \mathsf{Maybe}\ \iota)\ \tau \to \mathsf{R}\ a\ (\iota \to \tau) \end{array}$$

The constraint Eq will be used for handling non-linear uses of variables.

We also provide functions that manipulate these datatypes. It is convenient to have a map function for R a τ.

$$\mathit{rmap} :: (a \to b) \to \mathsf{R}\ a\ \tau \to \mathsf{R}\ b\ \tau$$

We omit the definition of *rmap*, which is straightforward. The function *upd* tries to update a given position in an environment.

$$\begin{array}{l} \mathit{upd} :: \mathsf{Eq}\ \iota \Rightarrow \mathsf{In}\ (\mathsf{Maybe}\ \iota)\ \Gamma \to \mathsf{Maybe}\ \iota \to \Gamma \to \Gamma \\ \mathit{upd}\ \mathsf{Z}\qquad a\ (\theta, a') = (\theta, a \oplus a') \\ \mathit{upd}\ (\mathsf{S}\ n)\ a\ (\theta, b)\ \ = (\mathit{upd}\ n\ a\ \theta, b) \end{array}$$

Here, $\oplus$ is the merging function defined as:

$$\begin{array}{l} (\oplus) :: \mathsf{Eq}\ a \Rightarrow \mathsf{Maybe}\ a \to \mathsf{Maybe}\ a \to \mathsf{Maybe}\ a \\ \mathsf{Nothing} \oplus b \qquad\qquad\quad = b \end{array}$$

$$
\begin{aligned}
&[\![\Gamma \vdash \lambda x.e : \iota \to \tau]\!] &&= \mathit{fabs}\ [\![\Gamma, x : \iota \vdash e : \tau]\!] \\
&[\![\Gamma \vdash e\ x : \tau]\!] &&= \mathit{fapp}\ [\![\Gamma \vdash e : \iota \to \tau]\!]\ [\![\Gamma(x) = \iota]\!] \\
&[\![\Gamma \vdash \mathit{text}\ s : \mathsf{D}]\!] &&= \mathit{ftext}\ s \\
&[\![\Gamma \vdash e_1 \mathbin{<>} e_2 : \mathsf{D}]\!] &&= \mathit{fcat}\ [\![\Gamma \vdash e_1 : \mathsf{D}]\!]\ [\![\Gamma \vdash e_2 : \mathsf{D}]\!] \\
&[\![\Gamma \vdash e_1 \mathbin{<?} e_2 : \mathsf{D}]\!] &&= \mathit{fchoice}\ [\![\Gamma \vdash e_1 : \mathsf{D}]\!]\ [\![\Gamma \vdash e_2 : \mathsf{D}]\!] \\
&[\![\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \{(\phi_i {\to} x_i) \to e_i\}_i]\!] = \\
&\qquad \mathit{fcase}\ [\![\Gamma(x) = \iota]\!]\ [\mathit{br}\ \phi_i\ [\![\Gamma, x_i : \iota' \vdash e_i : \tau]\!]]_i \\
&[\![\Gamma \vdash \mathbf{let}\ () = x\ \mathbf{in}\ e : \tau]\!] &&= \mathit{fununit}\ [\![\Gamma(x) = ()]\!]\ [\![\Gamma \vdash e : \tau]\!] \\
&[\![\Gamma \vdash \mathbf{let}\ (x_1, x_2) = x\ \mathbf{in}\ e : \tau]\!] &&= \\
&\qquad \mathit{funpair}\ [\![\Gamma(x) = (\iota_1, \iota_2)]\!]\ [\![\Gamma, x_1 : \iota_1, x_2 : \iota_2 \vdash e : \tau]\!]
\end{aligned}
$$

Figure 6. Translation of Terms-in-Context

$$
\begin{aligned}
&\mathsf{Just}\ a \quad \oplus \mathsf{Nothing} &&= \mathsf{Just}\ a \\
&\mathsf{Just}\ a \quad \oplus \mathsf{Just}\ a' \mid a == a' &&= \mathsf{Just}\ a
\end{aligned}
$$

Intuitively, $\mathit{upd}\ x\ a\ \theta$ computes $\theta \cup \{x \mapsto a\}$, which fails when θ maps x to some value other than a. The merging function can be extended to environments $\mathit{mergeEnv}_{[\![\Gamma]\!]} :: [\![\Gamma]\!] \to [\![\Gamma]\!] \to [\![\Gamma]\!]$, and then to parsing results $\mathit{merge}_{[\![\Gamma]\!]} :: \mathsf{R}\ [\![\Gamma]\!]\ \mathsf{D} \to \mathsf{R}\ [\![\Gamma]\!]\ \mathsf{D} \to \mathsf{R}\ [\![\Gamma]\!]\ \mathsf{D}$. Intuitively, $\mathit{mergeEnv}_\Gamma\ \theta\ \theta'$ represents $\theta \cup \theta'$, which fails when $\theta(x) \neq \theta'(x)$ for some x. These merging functions are type-indexed and are not directly expressible in Haskell, but as we have seen in Section 2 the solution is to use the unembedding technique of passing the value-level representation of Γ as a parameter. We also assume the typed-indexed function $\mathit{emptyEnv}_{[\![\Gamma]\!]} :: [\![\Gamma]\!]$ which denotes the environment that consists only of Nothing.

In advance to defining the translation of terms-in-context, we define the translation $[\![\Gamma(x) = \iota]\!] :: \mathsf{In}\ (\mathsf{Maybe}\ \iota)\ [\![\Gamma]\!]$ of variable look-up as below.

$$
\begin{aligned}
[\![(\Gamma, y : \iota')(x) = \iota]\!] &= \mathsf{S}\ [\![\Gamma(x) = \iota]\!] \\
[\![(\Gamma, x : \iota)(x) = \iota]\!] &= \mathsf{Z}
\end{aligned}
$$

For example, for $\Gamma = [\![x_1 : \iota_1, \dots, x_n : \iota_n]\!]$, we have $[\![\Gamma(x_i) = \iota_i]\!] = \mathsf{S}^{n-i+1}\ \mathsf{Z}$. That is, a variable is translated to a de Bruijn index.

Now, we are ready to define the translation of terms-in-context as in Fig. 6. To emphasize the compositionality of the definition, the translation uses the Haskell functions *fabs*, *fapp*, ..., *funpair* given in Fig. 7, which can be seen as a shallowly embedded version of de Brujin representation of the new FliPpr, and will be used in the unembedding. The definitions look complicated due to the manipulation of $\mathsf{R}\ \Gamma\ \tau$ values, but actually implement the same translation as shown in Section 2.1.2, except that function definitions and calls are separated into smaller steps. Note that branching is implemented by *fcase*, conversions of input data is by *br*, and *fununit* and *funpair* are responsible for data decomposition.

3.2 Embedded Non-Recursive FliPpr

With the ground prepared, the embedding itself is rather straightforward. We simply represent the syntax in the tagless-final style and then converts it to de Bruijn terms. Here, we

```
fabs :: Eq ι ⇒ Parser (R (Γ, Maybe ι) τ) → Parser (R Γ (ι → τ))
fabs p = ResF <$> p

fapp :: Eq ι ⇒ Parser (R Γ (ι → τ)) → In (Maybe ι) Γ →
          Parser (R Γ τ)
fapp p x = (λ(ResF r) → rmap (λ(θ, a) → upd x a θ) r) <$> p

ftext :: Parser (R Γ D)
ftext s = const (ResD emptyEnv_Γ) <$> ptext s

fcat :: Parser (R Γ D) → Parser (R Γ D) → Parser (R Γ D)
fcat p1 p2 = merge_Γ <$> p1 <*> p2

fchoice :: Parser (R Γ D) → Parser (R Γ D) → Parser (R Γ D)
fchoice p1 p2 = p1 <|> p2

fcase :: Eq ι ⇒ In (Maybe ι) Γ →
          [Parser (R (Γ, Maybe ι) D)] → Parser (R Γ D)
fcase x ps = rmap (λ(θ, a) → upd x a θ) <$> foldr (<|>) pfail ps
br :: Eq ι ⇒ PartialInj ι ι' →
        Parser (R (Γ, Maybe ι') D) → Parser (R (Γ, Maybe ι) D)
br (_, h) p = rmap (λ(θ, Just a) → (θ, Just (h a))) <$> p

fununit :: Eq ι ⇒ In (Maybe ι) Γ → Parser (R Γ τ)
fununit x p = rmap (upd x ()) <$> p
funpair :: (Eq ι1, Eq ι2) ⇒ In (Maybe (ι1, ι2)) Γ →
              Parser (R ((Γ, Maybe ι1), Maybe ι2) τ) → Parser (R Γ τ)
funpair x p =
  rmap (λ((θ, Just a), Just b) → upd x (Just (a, b))) <$> p
```

Figure 7. Semantics of Constructs as Haskell Functions

use shallow-embedding instead of ASTs in a datatype for the representation of de Bruijn terms.

3.2.1 Typeclass FliPprE

The following is the type class that represents the syntax of non-recursive FliPpr in the tagless-final style.

```
class FliPprE a e | e → a where
  abs :: Eq ι ⇒ (a ι → e τ) → e (ι → τ)
  app :: Eq ι ⇒ e (ι → τ) → a ι → e τ
  text :: String → e D
  (<>) :: e D → e D → e D
  (<?) :: e D → e D → e D
  case_ :: Eq ι ⇒ a ι → [Branch a e ι τ] → e τ
  unpair :: (Eq ι1, Eq ι2) ⇒
      a (ι1, ι2) → (a ι1 → a ι2 → e τ) → e τ
  ununit :: a () → e τ → e τ
data Branch a e ι τ where
  ∀ι'.Eq ι' ⇒ Branch (PartialInj ι ι') (a ι' → e τ)
```

Since FliPpr has two syntactic categories: variables and expressions, the class FliPprE a e takes two type variables a and e, respectively. The code is similar to the syntax in Section 3.1.1 except that we used functions for binders.

3.2.2 Instance of FliPprE for Parsing

We then implement the semantics of FliPpr by giving instances of FliPprE. Here, we focus on the parsing semantics, as the implementation of the pretty-printing semantics is straightforward.

First, we prepare the datatypes to which a and e of FliPprE $a\ e$ are instantiated to. Recall that a variable look-up and an expression (in a context) are translated to values in In (Maybe ι) Γ and Parser (R Γ τ), respectively. Accordingly, a and e will be instantiated to the following datatypes.

$$\textbf{data}\ \mathsf{PA}\ \iota = \mathsf{PA}\ \{\mathit{unPA} :: \forall \Gamma. \mathsf{TEnv}\ \Gamma \to \mathsf{In}\ (\mathsf{Maybe}\ \iota)\ \Gamma\}$$
$$\textbf{data}\ \mathsf{PE}\ \tau = \mathsf{PE}\ \{\mathit{unPE} :: \forall \Gamma. \mathsf{TEnv}\ \Gamma \to \mathsf{Parser}\ (\mathsf{R}\ \Gamma\ \tau)\}$$

Similarly to the original unembedding (Section 2.2), these types take value-level representations of Γ, TEnv Γ. A subtle difference is that we use shallow embedding instead of datatypes for de Bruijn terms. Also, since elements of Γ can be Nothing, we have changed the definition of TEnv as:

```
data TEnv γ where
  TEmp :: TEnv ()
  TExt :: Eq ι ⇒ TEnv r → Proxy ι → TEnv (r, Maybe i)
```

Then, we implement the method of FliPprE step-by-step. Again, *app* is rather easy to implement; we just pass γ around.

```
instance FliPprE PA PE where
  app :: PE (ι → τ) → PA ι → PE τ
  app (PE f) (PA a) = PE $ λγ → fapp (f γ) (a γ)
```

Notice that we use shallowly embedded construct *fapp* instead of a constructor. This applies to the implementation of other methods as well, such as *abs*, as below.

```
  abs :: (PA ι → PE τ) → PE (ι → τ)
  abs f = PE $ λγ →
    let γ_ι = TExt γ Proxy
    in fabs $ unPE (f (PA $ λγ' → coer γ_ι γ' Z)) γ_ι
```

One would notice that we simply replaced Abs in Section 2.2 by its semantics *fabs* in the above program.

The implementation of Wadler's combinators and nondeterministic choice is easy, as it does not involve binders.

```
  text :: String → PE D
  text s = PE $ λγ → ftext γ s
  (<>) :: PE D → PE D → PE D
  PE p1 <> PE p2 = PE (λγ → fcat γ (p1 γ) (p2 γ))
  (<?) :: PE D → PE D → PE D
  (PE p1) <? (PE p2) = PE (λγ → fchoice (p1 γ) (p2 γ))
```

Notice that now *ftext* and *fcat* take γ for type-indexed functions *emptyEnv* and *merge* that are used inside.

The *case_* method is implemented by *fcase* and *br* as below.

```
  case_ :: Eq ι ⇒ PA ι → [Branch PA PE ι τ] → PE τ
  case_ (PA a) bs = PE $ λγ →
    let h (Branch φ f) =
          let γ_ι' = TExt γ Proxy
              x = PA $ λγ' → coer γ_ι' γ' Z
          in br φ (unPE (f x) γ_ι')
    in fcase (a γ) (map h bs)
```

Notice that $\gamma_{\iota'}$ has type TEnv (Γ, Maybe ι'), where Γ and ι' come from γ :: TEnv Γ and ϕ :: PartialInj ι ι'.

The implementation of *ununit* is also straightforward as it does not change the type environment.

```
  ununit :: PA () → PE τ → PE τ
  ununit (PA a) (PE e) = PE $ λγ → fununit (a γ) (e γ)
```

In contrast, we need to use coercions in *unpair* as it involves binders.

```
  unpair :: (Eq ι1, Eq ι2) ⇒
              PA (ι1, ι2) → (PA ι1 → PA ι2 → PE τ) → PE τ
  unpair (PA a) k = PE $ λγ →
    let γ2 = TExt (TExt γ Proxy) Proxy
        x1 = PA $ λγ' → coer γ2 γ' (S Z)
        x2 = PA $ λγ' → coer γ2 γ' Z
    in funpair $ unPE (k x1 x2) γ2
```

Both functions just call corresponding implementations *fununit* and *funpair*, but the latter involves coercions.

3.3 Programming with FliPprE

Using raw *unpair*/*ununit* with Branch is sometimes tedious as they are too primitive. Haskell programming actually helps in this situation. For example, let us consider the subtraction language (Section 2.1.1) again. Assume that it is defined by the following datatype.

```
data Exp = One | Sub Exp Exp
```

Then, we can define the following functions.

```
unOne :: FliPprE a e ⇒ e t → Branch a e Exp t
unOne e = Branch (p, λ() → One) (λa → ununit a e)
  where p One = Just ()
        p _   = Nothing

unSub :: FliPprE a e ⇒
           (a Exp → a Exp → e t) ⇒ Branch a e Exp t
unSub k = Branch (p, q) (λx → unpair x k)
  where p (Sub x y) = Just (x, y)
        p _         = Nothing
        q (x, y)    = Sub x y
```

These functions serve as invertible pattern matching for better programming. For example, a prefix-notation printer for Exp can be defined as below.

```
prefix :: FliPprE a e ⇒ a Exp → e D
prefix x = case_ x
  [unOne $ text "1",
   unSub $ λx y → text "-" <> prefix x <> prefix y]
```

Here, we used Haskell recursions, which is enough for LL grammars and certain parser combinators such as `parsec`.

4 Embedding Recursive Definitions

Using Haskell-level recursions is nice, but it severely limits the expressive power. For example, we cannot express pretty-printers that are converted to left-recursive grammars (such as Fig. 2 and 3); parser combinators without explicit handling of recursions loop for them. Thus, we need to treat recursions explicitly so that we can generate arbitrary CFGs with conversions or analysis on them.

One natural solution would be having a fixed-point combinator. This would be achieved by adding a method $\mathit{fix} :: (e\ \tau \to e\ \tau) \to e\ \tau$ to the class $\mathsf{FliPpr}\ a\ e$. This solution works, but is unsatisfactory. The method itself does not provide a way to *share* generated sub-grammars, and will result in grammar-size blow-up for mutual recursions. We could use a variant that supports mutual recursions like $\mathit{fix} :: \mathit{Functor2}\ t \Rightarrow (t\ e \to t\ e) \to t\ e$, but still using fixed-point combinators prevents access to Haskell's syntactic support for defining recursions.

Thus, we resort to marking where recursions occurs, following `Earley`[2] and Frost et al. [11]'s parser combinators. This still allows us to define recursions by using Haskell's syntactic support under the `RecursiveDo` extension. Though this means that programmers now have the requirement of marking recursions, we believe it is not an onerous task.

Specifically, we use the following methods for marking.

```
class (FliPprE a e, MonadFix m) ⇒
        FliPprD m a e | e → a, e → m where
  mark :: e τ → m (e τ)
  local :: m (e τ) → e τ
```

The method *mark* marks recursive definitions. For example, *nil* and *space* in Section 2.1.1 will be implemented as below.

```
mkNilSp :: FliPprE m a e ⇒ m (e D, e D)
mkNilSp = do let white = text "  " <? text "\n"
             rec nil ← mark $ text "" <? space
                 space ← mark $ white <> nil
             return (nil, space)
```

We will use *mkNilSp* as **do** $\{(\mathit{nil}, \mathit{space}) \leftarrow \mathit{mkNilSp}; \dots\}$, where *mark* together with the monad ensures that *nil* and *space* will be shared; that is, nonterminals will be generated for *nil* and *space*, which will be used where we use *nil* and *space*, instead of copying their definitions.

The function *local* does the opposite; it cancels *mark* to convert sharable objects to unsharable ones. This is useful when we define recursions parameterized by other pretty-printing results, like *manyParens* function in Section 2 as below.

```
manyParens :: FliPprD m a e ⇒ e D → e D
manyParens d = local $ do
  rec x ← mark $ d <? text "(" <> nil <> x <> nil <> text ")"
  return x
```

Here, we assumed that *nil* appears in a context. Notice that it does not make sense to share *manyParens d* as it must yield different grammars for *d*. The use of *local* is also prompted by static typing, as without it *manyParens* will end up with a monadic type as a result of *mark*.

4.1 Representation of Grammars

Similar to the previous section, we focus on parsing semantics. Since now we need to generate recursive grammars, we need to specify how we represent them. For simplicity we shall use references provided by ST s monad; it also has the added benefit of working well with the *mark*ing approach,

```
data Grammar s a = G {unG :: ST s (StParser s a)}
```

The datatypes Grammar s and StParser s are assumed to share the same APIs (i.e., *ptext*, (<$>), (<*>), *pfail* and (<|>)) with Parser. A main difference from Parser is that the datetypes additionally have the following API for recursive definitions.

```
nt :: STRef s (ST s (StParser s a)) → StParser s a
```

Intuitively, *nt ref* represents a non-terminal, where *ref* points to its definition.

One of the uses of *nt* is to represent sharing.

```
gmark :: Grammar s a → ST s (Grammar s a)
gmark (G m) = do ref ← newSTRef m
                 return $ G (return (nt ref))
```

The *gmark* can be used to construct recursive grammars via MonadFix operations.

```
as :: ST s (Grammar s String)
as = do rec x ← gmark $ (ptext "") <|> ((++) <$> ptext "a" <*> x)
        return x
```

The argument of *nt* is a reference to a monadic computation instead of a pure expression, which essentially represents laziness. This is not so useful for now, but will be when we manipulate grammars, where we want also to delay monadic computation such as dereferencing.

4.2 Instance of FliPprD for Parsing

The changes to the underlying parser means that the PE type in Section 3.2.2 needs to be adapted; it now takes an additional type parameter s and uses Grammar s (R Γ τ) instead of Parser (R Γ τ). The rest of the code remains unchanged because Grammar s a and Parser a share the same APIs.

Now, we are ready to give a parsing instance. The first step is to prepare the following monad.

```
data PM s a = PM (∀Γ.TEnv Γ → ST s a)
```

[2] https://hackage.haskell.org/package/Earley

Notice that the above datatype is essentially a composition of Reader and ST monads with universal quantification on the reader argument. We omit the Functor, Applicative, Monad and MonadFix implementation of this datatype as they are standard. The TEnv Γ part will be used for communication between *local* and *mark*; *local* captures TEnv Γ and *mark* uses it. We also prepare the following datatype and function for this communication.

Then, we give a concrete instance of FliPprD, as below.

```
instance FliPprD (PM s) PA (PE s) where
  mark :: PE s τ → PM s (PE s τ)
  mark (PE e) = do
    SomeRep γ ← askTEnv
    g ← PM $ λ_ → gmark (e γ)
    return $ PE $ λγ′ → rmap (embedEnv γ γ′) <$> g
  local :: PM s (PE s τ) → PE s τ
  local (PM m) = PE $ λγ → G $
    m γ >>= λe → unG (unPE e γ)
```

Here, SomeRep and *askTEnv* are used for controlling type inference.

```
data SomeTEnv = ∀Γ.SomeTEnv (TEnv Γ)
askTEnv :: PM s SomeTEnv
askTEnv = PM (λγ → return $ SomeTEnv γ)
```

The function *embedEnv* :: TEnv Γ → TEnv Γ′ → Γ → Γ′ converts environments by adding Nothing to the right. The idea of *local* is to capture the value-level type environment of where *local* is called. The captured type environment γ will be used by *mark* e to evaluate e, and the marked result will be used under a deeper context γ'.

Similarly to *coer*, we expect Γ' being at least as big as Γ (or, Γ is a sub-environment of Γ'). Unfortunately, this property is not guaranteed by Atkey et al. [2]'s unembedding, but we believe that it holds as *mark*ed recursions only occur inside *local*. Note that to use the *mark*ed functions outside *local*, they have to be put as the return value of the argument of *local* such as *local* (**do** {**rec** x ← *mark* ...; *return* x}). We believe that this property could be proved by a similar discussion to Atkey [1], which is left for future work.

Finally, we define the parsing interpretation as below.

```
parser :: (∀m a e.FliPprD m a e ⇒ m (e (ι → D)))
         → (∀s.Grammar s ι)
parser (PM m) = G $ do e ← m TEmp
                       unG (f <$> unPE e TEmp)
    where f :: R () (ι → D) → ι
          f (ResF (ResD (_, Just a))) = a
```

5 Further Improvements

We discuss several improvements of the basic embedded implementation of FliPpr, from both programming and efficiency perspectives.

5.1 Wrapping Raw Type Variables

The current APIs of the embedded FliPpr expose raw type variables a and e. This is inconvenient if we want to make FliPpr syntax as an instance of a type class. For example, we may want to use the same APIs for both FliPpr programs and the usual pretty-printing.

Let us assume that the APIs developed so far are located under a module Core. Then, we provide a "wrapped" version of the APIs as below.

```
newtype A a ι = A {unA :: a ι}
newtype E a τ = E {unE :: a τ}
abs :: (FliPprE a e, Eq ι) ⇒ (A a ι → E e τ) → E e (ι → τ)
abs f = E (Core.abs (unE ∘ f ∘ A))
...
```

With this wrapped APIs, we can make instances without worrying about overlapping instances. For example, we can make FliPpr programs as a Monoid instance.

```
instance (τ ∼ D) ⇒ Monoid (E e τ) where
  mempty = text ""
  mappend (E e1) (E e2) = E (e1 Core.<> e2)
```

Here, the constraint $\tau \sim$ D saves us from cluttering the constraints Monoid (E e τ) for uses of *mempty* and *mappend*.

5.2 Inter-conversion from/to Haskell Functions

Using *app* and *abs* explicitly for every function definition and application is tedious. To resolve the issue, we provide the following type class for inter-conversion between E e $(\iota_1 \to \cdots \to \iota_n \to$ D$)$ and A a $\iota_1 \to \cdots \to$ A a $\iota_n \to$ E e D, by using *abs* and *app*.

```
class Repr (a :: ∗ → ∗) e τ r
                | e → a, e τ → r, r → a e τ where
  toFunction :: E e τ → r
  fromFunction :: r → E e τ
```

The type class has the following instances.

```
instance FliPprE a e ⇒ Repr a e D (E e D) where ...
instance (FliPprE a e, Repr a e τ r, Eq ι) ⇒
           Repr a e (ι → τ) (A a ι → r) where ...
```

We omit the definitions of *toFunction* and *fromFunction*, which follow straightforwardly from their types.

With this type class, we can define the following function.

```
define :: (FliPprD m a e, Repr a e τ r) ⇒ r → m r
define f = fmap toFunction $ mark (fromFunction f)
```

Function *define* eliminates direct use of *app* and *abs*. Now we can write

```
do rec f ← define $ λx → ... f x ...
     ... f y ...
```

instead of:

```
pprMain :: FliPprD a e ⇒ A a Exp → E e D
pprMain = do
  (nil, space) ← mkNilSp
  spaceN ← define $ space <? text ""
  lineN ← line <? text ""
  let parens d = text "(" <> nil <> x <> nil <> text ")"
  let parensIf b d = if b then parens d else d
  let manyParens d = local $ do rec x ← d <? parens x
                                    return x
  rec ppr ← defines [False, True] $ λb x → manyParens $
        parensIf b $ case_ x
          [unOne $ text "1",
           unSub $ λx y →
             ppr False x <>
             nest 2 (lineN <> text "-" <> spaceN <> ppr True y)]
  return $ ppr False
```

Figure 8. An Embedded FliPpr Program Equivalent to Fig. 1

```
do rec f ← mark $ abs $ λx → ... app f x ...
     ... app f y ...
```

5.3 Parameterized Recursions

When writing pretty-printers, we often pass a precedence level of a context to decide whether a pretty-printer produces a pair of opening and closing parentheses. For the simple subtraction language, there are only two precedence levels, and thus we pass booleans in Fig. 1. This way of handling precedence is not directly allowed by *mark* or *define*.

Thus, we define *defines* as below.

```
defines :: (Eq k, Ord k, FliPprD m a e, Repr a e τ r) ⇒
             [k] → (k → r) → m (k → r)
defines ks f = do
  rs ← mapM (define ∘ f) ks
  let tab = Data.Map.fromList $ zip ks rs
  return $ λk → fromJust $ (Data.Map.lookup k tab)
```

The function *fromJust* in `Data.Maybe` removes Just, which fails if the input is Nothing. The definition might look complicated, but *defines* $[k_1, \ldots, k_n]$ f essentially *defines* each $f\ k$, and makes a table for looking-up a defined function.

As a result, we can write the pretty-printer for the simple subtraction language as Fig. 8.

5.4 Special Treatment of Spacing Combinators

One may find it tedious to copy the definitions of *nil*, *space*, *spaceN* and *lineN* to every pretty-printing definition. We may apply *local* to these functions, but then the grammars generated by the combinators are no longer shared.

Thus we include them to the FliPpr APIs, i.e., FliPprE's class methods. This is also useful when we parse languages that support comment syntax, where we want spacing combinators to in addition skip comments in parsing. By including them in the APIs, *white* can be specified at parser generation time, making invertible pretty-printing combinators like *manyParens* more reusable.

5.5 Implementation of Type/Variable Environments

The value-level type TEnv is represented as a list-like structure, and as a result the coercion *coer* $\gamma\ \gamma'$ takes time quadratic to the size of γ, which is unacceptable. Another source of inefficiency is the representation of value environments; *mergeEnv* $\gamma\ \theta\ \theta'$ takes time linear to the size of $\gamma/\theta/\theta'$, while most of elements in θ and θ' are often Nothing.

To avoid these overhead, we just pass the size of γ (i.e., the nesting depth of binders) and use *unsafeCoerce* if needed. We also change the representation of value environments so that a consecutive block of elements can be Nothing. This makes *coer*, *mergeEnv*, *emptyEnv* and *embedEnv* efficient. The function *upd* n still takes time linear to n, but it is less problematic as n tends to concern recently introduced variables and therefore is usually small.

6 A Larger Example

In this section, we demonstrate the programmability of embedded FliPpr by defining an invertible pretty-printer for the following AST. As we will see, the embedding not only have preserved the benefits of FliPpr, but also enhanced its programmability through the interaction with the host language. Reference code for the original FliPpr version and non-invertible pretty-printer version can be found in Appendix A.2 for comparison.

```
data Exp = Num Int | Var String | Let String Exp Exp
         | Sub Exp Exp | Div Exp Exp
```

Despite being simple, the above expression language contains common features in programming languages: keywords, constants and operators with precedence. We assume decomposing functions such as *unSub* for the constructors. Our current implementation uses Template Haskell to generate such functions.

Let us consider constants and variables. In the original FliPpr, this is done by using the *text* $(f\ x)$ **as** r expression that pretty-prints $f\ x$ and parses the regular expression r with conversion f^{-1}, for an injection f. For example, an integer n is printed by *text* (*itoa* n) **as** `-?[0-9]+` and variable x is printed by *text* x **as** `[a-z][a-zA-Z0-9]*-let`, where `-` outside of square brackets represents subtraction.

So our first goal is to give an equivalent expression in the embedded FliPpr. First, we prepare a function that makes a printer from a deterministic finite-state automaton (DFA).

```
type Q = Int
data DFA = DFA Q [(Q, [(Char, Q)])] [Q]
fromDFA :: FliPprD m a e ⇒ DFA → m (A a String → E e D)
```

```
fromDFA (DFA init tr fs) = do
  rec abort ← define abort
  rec f ← defines (map fst tr) $ λq s → case_ s $
       [unNil $ (if elem q fs then text "" else abort),
        unCons $ λa r → case_ a
          [is c $ text [c] <> (f q′ r) |
             (c, q′) ← fromJust (lookup q tr)]]
  return (f init)
```

The function *is*, which works as an invertible constant pattern, is defined as below.

```
is :: (Eq ι, FliPprE a e) ⇒ ι → E e t → Branch (A a) (E e) ι t
is c = Branch (p, const c) (λa → ununit a e)
  where p x = if x == c then Just () else Nothing
```

Assuming that we already have DFAs dfa_{num} and dfa_{var} for integers and variable names, respectively, then we can make a function for generating printers.

```
mkPprInt :: FliPprD m a e ⇒ m (A a Int → E e D)
mkPprInt = do f ← fromDFA dfa_num
              retrun $ λx → case_ x [itoa $ f]
mkPprVar :: FliPprD m a e ⇒ m (A a String → E e D)
mkPprVar = fromDFA dfa_var
```

Here, *itoa* is defined by *itoa* = Branch (Just ∘ *show*, *read*). Those functions will be used as *pprInt* ← *mkPprInt* to avoid duplicating nonterminals to parse integers or variables.

Next, we prepare a template for pretty-printers of arithmetic expressions.

```
type Prec = Int
data Assoc = AL | AR | AN
data Fixity = Fixity Prec Assoc
opP :: FliPpr a e ⇒ Fixity → (E e τ → E e τ → E e τ) →
       (Prec → A a ι1 → E e τ) → (Prec → A a ι2 → E e τ) →
       Prec → A a ι1 → A a ι2 → E e τ
opP (Fixity k opPrec) f p1 p2 k x y =
  let (d1, d2) = case a of {AL → (0, 1); AR → (1, 0); _ → (0, 0)}
  in parensIf (k > opPrec) $
       f (p1 (opPrec + d1) x) (p2 (opPrec + d2) y)
```

Now, we are ready to define a pretty-printing function for the language.

```
ppr :: FliPprD m a e ⇒ m (A a Exp → E e D)
ppr = do
  pprInt ← mkPprInt
  pprVar ← mkPprVar
  let op s d1 d2 = group $
        d1 <> nest 2 (lineN <> text s <> spaceN <> d2)
  rec pprE ← defines [0..3] $ λk e → manyParens $ case_ e $
        [unNum $ pprInt,
         unVar $ pprVar,
         unSub $ opP (Fixity 1 AL) (op "-") pprE pprE k,
         unDiv $ opP (Fixity 2 AL) (op "/") pprE pprE k,
         unLet $ λx e1 e2 → parensIf (k > 0) $ group $
           text "let" <+> pprVar x <> nil <> text "=" <>
             nest 2 (lineN <> pprE 0 e1) <>
           line <> text "in" <+> pprE 0 e2]
  return (λx → nil <> pprE 0 x <> nil)
    where x <+> y = x <> space <> y
```

As demonstrated in the above example, we can use Haskell functions (such as *fromDFA*), including higher-order ones (such as *opP* and *is*), to build FliPpr programs. This is not possible in the original FliPpr, except some special cases with the designated syntax *text s* **as** *r* (see Appendix A.2 for a comparison).

7 Discussions and Related Work

In this paper, we looked at the embedding of FliPpr. Through the techniques are presented in the specific context, some of the results are expected to be of more general interests.

Recall that there are three characteristics of FliPpr language: (1) treelessness, (2) first-orderness and (3) explicit handling of recursions. These characteristics are actually rather common in invertible languages. For example, Matsuda et al. [23] and Nishida et al. [29] also discuss the inversion of treeless languages. Since treelessness essentially characterizes transducer-like computation, where the inputs and outputs are separated, we believe a similar technique would be applicable to invertible transducers [15, 22].

Not all languages require an elaborate treatment of recursions like the case of FliPpr. For lenses [10] and reversible functional languages such as RFUN [38], the usual (i.e., Haskell-level) fixed-point is sufficient. Consequently, there is no need to have *abs* and *app*, as using Haskell-level functions on the guest-language's expressions would suffice. However, there are still other binders (such as **let** and **case** expressions) where the unembedding transformation is needed.

Some program inversion methods are realized by whole program analysis and transformation [12]. That is, they are transformations from a programming language to another, which is similar to the original spirit of the unembedding.

There are other embedded systems for defining pairs of parser and printer. Rendel and Ostermann [34] propose an embedded invertible syntax description framework based on arrow combinators [17], in which users define printers and parsers in the same language. But they do not support control on pretty-printing (i.e., *group* and *nest*), nor point-wise programming. Despite being based on arrow combinators, invertible syntax description is not proper arrows and thus not subject to the arrow syntax [32]. Moreover, the framework is hardwired to a certain parsing semantics, whereas ours generates CFGs open to different parsing algorithms. Danielsson [7] develops a framework in Agda, in which users write a grammar and a pretty-printer, where the correctness of the pretty-printer with respect to the grammar is guaranteed by construction with the help of dependent types, as the pretty-printing combinators convey proofs. Unlike ours and the invertible syntax framework, users write both a pretty-printer and a grammar, which leads to a maintenance problem: changes to one may imply non-trivial changes to the other.

We use the ST monad for representing grammars, while Baars et al. [3] use de Bruijn index for type-safe representation of recursive grammars. To use de Bruijn index in our setting requires elaborating type-level programming to deal with mutually defined functions. We also note that the idea of using monads to express laziness and sharing can be found in Fischer et al. [9], and Matsuda and Asada [21].

Parametric higher-order abstract syntax (PHOAS) [5] is another technique for reusing host language's binders. This representation has the similar problem with the tagless-final style in embedding invertible languages. Moreover, *mark*-like methods that do not return the expression type are not well expressed in PHOAS.

Polakow [33] proposes an embedding method of the linear λ calculus to Haskell, which does not require explicit weakening of terms. Although there is no weakening in the linear λ calculus, he considers a variant of which typing judgment has the form of $\Gamma_1 \setminus \Gamma_2 \vdash e : \tau$, where the difference between Γ_1 and Γ_2 represents the original linear type environment, but allows weakening-like conversion from $\Gamma_1 \setminus \Gamma_2 \vdash e : \tau$ to $(\Gamma_1, \Gamma') \setminus (\Gamma_2, \Gamma') \vdash e : \tau$. He avoids explicit conversion by abstracting a type environment through polymorphism; type instantiation suffices for weakening because de Bruijn levels are used instead of indices. The technique is also useful for a non-linear setting as in FliPpr. However, being polymorphic complicates manipulation of terms. For example, explicit type signatures are mandatory for recursive definitions [14], while being optional in unembedding.

Matsuda and Wang [25, 26] provide a way to convert lenses [10] to functions via Yoneda embedding, which enables us to compose lenses via Haskell's usual higher-order functions. Since invertible functions are a special case of lenses, we could use this approach for pretty-printing primitives. However, the method does not handle lens combinators well and is not sufficient for our purpose. For example, there will be a restriction that case branches much be closed, ruling out programs such as *fromDFA*. The language HOBiT is designed [27] to overcome this problem. But just like FliPpr, HOBiT is standalone, which may also benefit from the techniques proposed in this paper for an embedded implementation.

8 Conclusion

We have developed an embedded version of FliPpr using the unembedding transformation [1, 2]. The benefit is enhanced interoperability with Haskell (as the host language): one can interconvert FliPpr functions and Haskell functions, and FliPpr functions can manipulate Haskell's datatypes. This newly gained power is useful. We are now able to construct FliPpr programs using Haskell functions, avoiding rather complex programs transformations and syntactic restrictions of the original FliPpr—they can be mimicked by the new APIs (*mark* and *local*) and Haskell function (*defines*).

A Appendix

A.1 Implementation of *coer*

The implementation is a bit different from the untyped case [2] and the Agda implementation case [19]. The basic structure of *coer* is as follows.

```
coer γ γ' x | Just Refl ← eqEnv γ γ' = x
coer γ (TExt γ' _)                   = S (coer γ γ')
```

Here, Refl is the constructor of the following datatype that represents propositional equality.

```
data a :~: b where Refl :: a :~: a
```

There are two ways to implement *eqEnv*. One approach is to use *eqT* from `Data.Typeable`.

```
eqEnv :: (Typeable Γ, Typeable Γ') ⇒
           TEnv Γ → TEnv Γ' → Maybe (Γ :~: Γ')
eqEnv _ _ = eqT
```

This works and is efficient (*eqT* runs in O(1) time as it performs comparison (only) on hash values), but requires Γ and Γ' to be Typeable instances, scattering Typeable constraints to *coer* and the TEnv definition and so on.

Thus, for simplicity of presentation, we avoid the above definition and use the following definition instead.

```
eqEnv :: TEnv Γ → TEnv Γ' → Maybe (Γ :~: Γ')
eqEnv TEmp TEmp = Just Refl
eqEnv (TExt γ _) (TExt γ' _) =
  case eqEnv γ γ' of
    Nothing → Nothing
    Just Refl → Just (unsafeCoerce Refl)
eqEnv _ _ = Nothing
```

Notice that we have $\Gamma = \Gamma'$ if γ and γ' have the same size [1], and the use of *unsafeCoerce* does not risk type safety. This version of *coer* γ γ' takes time quadratic to the size of γ. As discussed in Section 5.5, we use a more efficient implementation with more aggressive use of *unsafeCoerce* to make *coer* constant time. The actual implementation can be found in the module `Text.FliPpr.Internal.PartialEnv` in the implementation site.

A.2 Code Comparison

The following is a program in the original FliPpr's surface language, which corresponds to Section 6.

```
ppr x = nil <> pprE 0 x <> nil
pprVar x = text x as ([a-z][a-zA-Z0-9]*)-let
pprE k x = manyParens (pprE' k x)
pprE' k (Num n) = text (itoa n) as -?[0-9]+
pprE' k (Var x) = pprVar x
```

```
pprE' k (Sub e1 e2) = ifParens (k > 1) (group (
  pprE k e1 <>
    nest 2 (lineN <> text "-" <> spaceN <> pprE 2 e2)))
pprE' k (Div e1 e2) = ifParens (k > 2) (group (
  pprE k e1 <>
    nest 2 (lineN <> text "/" <> spaceN <> pprE 3 e2)))
pprE' k (Let x e1 e2) = parensIf (k > 0) (group (
  text "let" <> space <> pprVar x <> nil <> text "=" <>
    nest 2 (lineN <> pprE 0 e1) <>
  line <> text "in" <> space <> pprE 0 e2))
```

One might find that Sub and Div branches are similar, but since the original FliPpr is first-order, we cannot extract the common pattern between the branches. A subtle difference is that we replaced <+> with its definition as the original FliPpr does not allow users to define binary operators.

If we just use Wadler's combinators in Haskell, the code would be as follows.

```
ppr :: Exp → Doc
ppr x = pprE 0 x
pprE :: Prec → Exp → Doc
pprE k (Num n) = text (show n)
pprE k (Var x) = text x
pprE k (Sub e1 e2) = opP (Fixity 1 AL) (op "-") pprE k e1 e2
pprE k (Div e1 e2) = opP (Fixity 2 AL) (op "/") pprE k e1 e2
pprE k (Let x e1 e2) = parensIf (k > 0) $ group $
  text "let" <+> text x <> text "=" <>
    nest 2 (line <> pprE 0 e1) <>
  line <> text "in" <+> pprE 0 e2
op :: String → Doc → Doc
op s d1 d2 = group $ d1 <> nest 2 (line <> text s <+> d2)
(<+>) :: Doc → Doc → Doc
x <+> y = x <> text " " <> y
opP :: Fixity → (Doc → Doc → Doc) →
        (Prec → a → Doc) → (Prec → b → Doc) →
        Prec → a → b → Doc
opP = ... {- the same definition as Section 6 -} ...
```

Here, we use Doc for the objects that retain pretty-printing information in Wadler's combinators [36]. Notice that we do not use *manyParens*, *nil*, *space* and *spaceN* here because we do not specify parsing behavior in pure pretty-printing. It is interesting see that the definition of *opP* is the same as that in Section 6, which is impossible in the original FliPpr. On the other hand, both original FliPpr and Haskell versions use ordinary pattern matching, which has to be simulated by deconstructing functions such as *unSub* in embedded FliPpr.

Acknowledgments

The work was partially supported by JSPS KAKENHI Grant Numbers 15K15966 and 15H02681, and by Royal Society International Exchanges Grant: *Bidirectional Compiler for Software Evolution*, IES\R3\170104.

References

[1] Robert Atkey. 2009. Syntax for Free: Representing Syntax with Binding Using Parametricity. In *TLCA (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, 35–49. https://doi.org/10.1007/978-3-642-02273-9_5

[2] Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Haskell*, Stephanie Weirich (Ed.). ACM, 37–48. https://doi.org/10.1145/1596638.1596644

[3] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. 2010. Typed Transformations of Typed Grammars: The Left Corner Transform. *Electr. Notes Theor. Comput. Sci.* 253, 7 (2010), 51–64. https://doi.org/10.1016/j.entcs.2010.08.031

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[5] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[7] Nils Anders Danielsson. 2013. Correct-by-construction pretty-printing. In *DTP*, Stephanie Weirich (Ed.). ACM, 1–12. https://doi.org/10.1145/2502409.2502410

[8] Jonas Duregård and Patrik Jansson. 2011. Embedded parser generators. In *Haskell*, Koen Claessen (Ed.). ACM, 107–117. https://doi.org/10.1145/2034675.2034689

[9] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely functional lazy nondeterministic programming. *J. Funct. Program.* 21, 4-5 (2011), 413–465. https://doi.org/10.1017/S0956796811000189

[10] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007).

[11] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *PADL (Lecture Notes in Computer Science)*, Paul Hudak and David Scott Warren (Eds.), Vol. 4902. Springer, 167–181.

[12] Robert Glück and Masahiko Kawabe. 2004. Derivation of Deterministic Inverse Programs Based on LR Parsing. In *FLOPS (Lecture Notes in Computer Science)*, Yukiyoshi Kameyama and Peter J. Stuckey (Eds.), Vol. 2998. Springer, 291–306.

[13] Robert Glück and Masahiko Kawabe. 2005. Revisiting an automatic program inverter for Lisp. *SIGPLAN Notices* 40, 5 (2005), 8–17.

[14] Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (1993), 253–289. https://doi.org/10.1145/169701.169692

[15] Qinheping Hu and Loris D'Antoni. 2017. Automatic program inversion using symbolic transducers. In *PLDI*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 376–389. https://doi.org/10.1145/3062341.3062345

[16] John Hughes. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming (Lecture Notes in Computer Science)*, Johan Jeuring and Erik Meijer (Eds.), Vol. 925. Springer, 53–96.

[17] John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[18] Graham Hutton. 1992. Higher-Order Functions for Parsing. *J. Funct. Program.* 2, 3 (1992), 323–343. https://doi.org/10.1017/S0956796800000411

[19] Steven Keuchel and Johan Jeuring. 2012. Generic conversions of abstract syntax representations. In *WGP*, Andres Löh and Ronald Garcia (Eds.). ACM, 57–68. https://doi.org/10.1145/2364394.2364403

[20] Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In *APLAS (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 357–376. https://doi.org/10.1007/978-3-319-47958-3_19

[21] Kazutaka Matsuda and Kazuyuki Asada. 2017. A functional reformulation of UnCAL graph-transformations: or, graph transformation as graph reduction. In *PEPM*, Ulrik Pagh Schultz and Jeremy Yallop (Eds.). ACM, 71–82. https://doi.org/10.1145/3018882

[22] Kazutaka Matsuda, Kazuhiro Inaba, and Keisuke Nakano. 2012. Polynomial-time inverse computation for accumulative functions with multiple data traversals. *Higher-Order and Symbolic Computation* 25, 1 (2012), 3–38. https://doi.org/10.1007/s10990-013-9097-8

[23] Kazutaka Matsuda, Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2010. A Grammar-Based Approach to Invertible Programs. In *ESOP (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 448–467.

[24] Kazutaka Matsuda and Meng Wang. 2013. FliPpr: A Prettier Invertible Printing System. In *ESOP (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 101–120. https://doi.org/10.1007/978-3-642-37036-6_6

[25] Kazutaka Matsuda and Meng Wang. 2015. Applicative bidirectional programming with lenses. In *ICFP*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 62–74. https://doi.org/10.1145/2784731.2784750

[26] Kazutaka Matsuda and Meng Wang. 2018. Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. *J. Funct. Program.* 28 (2018), e15. https://doi.org/10.1017/S0956796818000096

[27] Kazutaka Matsuda and Meng Wang. 2018. HOBiT: Programming Lenses Without Using Lens Combinators. In *ESOP (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 31–59. https://doi.org/10.1007/978-3-319-89884-1_2

[28] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[29] Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. 2001. Generation of inverse term rewriting systems for pure treeless functions. In *Proceedings of the International Workshop on Rewriting in Proof and Computation*. 188–198.

[30] Hugo Pacheco and Alcino Cunha. 2010. Generic Point-free Lenses. In *MPC (Lecture Notes in Computer Science)*, Claude Bolduc, Jules Desharnais, and Béchir Ktari (Eds.), Vol. 6120. Springer, 331–352. https://doi.org/10.1007/978-3-642-13321-3_19

[31] Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014. Monadic combinators for "Putback" style bidirectional programming. In *PEPM*, Wei-Ngan Chin and Jurriaan Hage (Eds.). ACM, 39–50. https://doi.org/10.1145/2543728.2543737

[32] Ross Paterson. 2001. A New Notation for Arrows. In *ICFP*, Benjamin C. Pierce (Ed.). ACM, 229–240. https://doi.org/10.1145/507635.507664

[33] Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Haskell*, Ben Lippmeier (Ed.). ACM, 177–188. https://doi.org/10.1145/2804302.2804309

[34] Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. In *Haskell*, Jeremy Gibbons (Ed.). ACM, 1–12.

[35] Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248.

[36] Philip Wadler. 2003. A Prettier Printer. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave Macmillan, Chapter 11.

[37] Tetsuo Yokoyama. 2010. Reversible Computation and Reversible Programming Languages. *Electr. Notes Theor. Comput. Sci.* 253, 6 (2010), 71–81. https://doi.org/10.1016/j.entcs.2010.02.007

[38] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2011. Towards a Reversible Functional Language. In *RC (Lecture Notes in Computer Science)*, Alexis De Vos and Robert Wille (Eds.), Vol. 7165. Springer, 14–29. https://doi.org/10.1007/978-3-642-29517-1_2