Embedding Invertible Languages with Binders
A Case of the FliPpr Language

Kazutaka Matsuda
Tohoku University

Meng Wang
University of Bristol

Haskell Symposium@Sep. 27, 2018
Motivation

How can we embed languages with *non-functional semantics*?

$[a \to b]$ is not a function from $[a]$ to $[b]$

Especially, those with *binders*?

functions, let-expressions, pattern matching, etc.
Language w/ Non-Functional Sem

- Invertible [Yokoyama+ 11, M+10,...]
  \[
  [a \rightarrow b] = ([a] \rightarrow [b], [b] \rightarrow [a])
  \]

- Bidirectional [Foster+05, 07...]
  \[
  [a \rightarrow b] = \text{Lens} [a] [b]
  = ([a] \rightarrow [b], [a] \rightarrow [b] \rightarrow [a])
  \]
  (NB: not-necessarily higher-order)
Aims (in terms of Invertible Lang)

- Express guest's binders by host's funcs.
  - HOAS (e.g. tagless-final style [Carette+09])

$$\text{class Lam e where} \quad \lambda x. e$$

\begin{align*}
\text{abs} & \::\ (e \sigma \rightarrow e \tau) \rightarrow e (\sigma \rightarrow \tau) \\
\text{app} & \::\ e (\sigma \rightarrow \tau) \rightarrow e \sigma \rightarrow e \tau
\end{align*}

- Implement inverse semantics

\[
\text{Inv}[[\Gamma \vdash e : \tau]] :: [[\tau]] \rightarrow [[\Gamma]]
\]

cf. \[
[[\Gamma \vdash e : \tau]] :: [[\Gamma]] \rightarrow [[\tau]]
\]
Issues

- No explicit environments in HOAS
  - NB: PHOAS has the same problem [Chripara08]

```haskell
class Lam e where
  abs :: (e σ -> e τ) -> e (σ -> τ)
  app :: e (σ -> τ) -> e σ -> e τ
```

- But, the semantics refers to envs

\[ Inv[[Γ ⊢ e : τ]] :: [[τ]] \rightarrow [[Γ]] \]

\[ cf. [[Γ ⊢ e : τ]] :: [[Γ]] \rightarrow [[τ]] \]
Approach

- Use *unembedding* [Atkey+09]

\[
\text{unembed} :: (\forall e. \text{Lam } e \Rightarrow e \tau) \rightarrow \text{DLam } () \tau
\]

```haskell
class Lam e where
  abs :: (e \sigma \rightarrow e \tau) \rightarrow e (\sigma \rightarrow \tau)
  app :: e (\sigma \rightarrow \tau) \rightarrow e \sigma \rightarrow e \tau
```

```haskell
data DLam env a where
  Var :: In \sigma env \rightarrow DLam env \sigma
  Abs :: DLam (env,\sigma) \tau \rightarrow DLam env (\sigma \rightarrow \tau)
  App :: DLam env (\sigma \rightarrow \tau) \rightarrow DLam env \sigma \rightarrow DLam env \tau
```

Access to env!
This Paper

- Embedding FliPpr [M&W13]
  - FliPpr: an *invertible* language
    - takes a pretty-printer
    - returns a corresponding parser
  - To achieve interoperability with Haskell
    - ASTs defined by Haskell datatypes
    - FliPpr programs generated by Haskell functions
Contributions

- Embedding invertible languages through unembedding
- Redesign of FliPpr to enhance interoperability with Haskell
- Discussions on treatment of rather complex features in FliPpr
Agenda

- Background: FliPpr
- Embedding FliPpr by Unembedding
  - Interoperable FliPpr
  - Handling Recursions
Pretty-Printers in the Core Language

Grammar-based inversion [M+10]

Context-Free Grammars with Actions
FliPpr System [M&W13]

Pretty-Printers in the Surface Language

partial evaluation & fusion

Pretty-Printers in the Core Language

Grammar-based inversion [M+10]

Context-Free Grammars with Actions
FliPpr Core

- **Treeless** [Wadler90] **1st order** language w/ pretty-printing combinators [Wadler03]

\[
\begin{align*}
\text{prog} & ::= r_1 \ldots r_n \\
r & ::= f(p_1, \ldots, p_n) = e \\
e & ::= op e_1 \ldots e_n \\
&\quad| f \ x_1 \ldots x_n
\end{align*}
\]

- Easy to invert, hard to program with
  - Inverses are in CFG with actions

Arguments must be variables (treeless)
FliPpr Surface

- Statically computed arguments
- Relaxed treelessness restriction
  - Each function has a tier. Functions in a tier are treated as combinators in later tiers

```haskell
ppr b x = manyParens (aux b x)
aux _ One = text "1"
aux b (Sub x y) = group (parensIf b (ppr False x <> nest 2 (line <> text "-" <> space <> ppr True y)))
manyParens d = d <? parens (manyParens d)
parensIf b d = if b then parens d else d
```
Pretty-Printing & Parsing

ppr b x = manyParens (aux b x)

aux _ One = text "1"

aux b (Sub x y) = group (parensIf b (ppr False x <> nest 2

(line <> text "-" <> space <> ppr True y)))

manyParens d = d <? parens (manyParens d)

parensIf b d = if b then parens d else d

Sub (Sub One One) One ➞ 1 – 1 – 1 1 – 1

- 1
Pretty-Printing & Parsing

\[
\begin{align*}
\text{ppr } b \ x &= \text{manyParens} \ (\text{aux } b \ x) \\
\text{aux } \_ \ One &= \text{text } "1" \\
\text{aux } b \ (\text{Sub } x \ y) &= \text{group} \ (\text{parensIf } b \ ( \\
  \text{ppr } \text{False} \ x \ \text{< <>nest} \ 2 \ \\
  (\text{line} \text{< >text } "-" \text{< >space} \text{< >ppr } \text{True} \ y))) \\
\text{manyParens} \ d &= d \ ? \ \text{parens} \ (\text{manyParens} \ d) \\
\text{parensIf } b \ d &= \text{if } b \ \text{then} \ \text{parens} \ d \ \text{else} \ d
\end{align*}
\]

Sub (Sub One One) One

\[
\begin{align*}
1 - 1 - 1 & \quad 1 - 1 \\
((1 - 1) - 1) & \quad \ldots
\end{align*}
\]
Why Embedding?

- Interoperability with Haskell
  - pretty printers/parsers for user-defined types with type checking
- Replace the surface language with (meta)programming in Haskell
  - to avoid complex implementations
- Type-based restrictions rather than syntactic restrictions
Agenda

- Background: FliPpr
- Embedding FliPpr by Unembedding
  - Interoperable FliPpr
  - Handling Recursions
Interoperable FliPpr

- Redesign of FliPpr Core
  - greater interoperability with Haskell
    - allows pretty-printers to manipulate user-defined Haskell's datatypes
  - use λs instead of global function defs
    - good for embedding
    - still first-order and treeless
**Syntax (w/o Recursion)**

\[ e ::= \lambda x.e \mid e\ x \mid op\ e_1 \ldots\ e_n \]

- case \( x \) of \( \{ (\phi_i \rightarrow x_i) \rightarrow e_i \} \)
- let () = \( x \) in \( e \) | let (\( x_1, x_2 \)) = \( x \) in \( e \)

*for pattern matching*

- \( op \): Wadler's combinators
- \( \phi_i \): Haskell-level partial injections

\[ \text{type PInj} \ s\ t = (s \rightarrow \text{Maybe } t, t \rightarrow s) \]

**NB:** the language is 1st-order and treeless
Type Class: FliPprE

class FliPprE a e where
   abs :: (a σ → e τ) → e (σ → τ)
   app :: e (σ → τ) → a σ → e τ
   case_ :: a σ → [Br a e σ τ] → e τ
   ununit :: a () → e τ → e τ
   unpair :: a (σ₁,σ₂) → (a σ₁ → a σ₂ → e τ) → e τ
   text :: String → e Doc

... pretty-printing result

data Br a e σ τ = ∀σ'. Br (PInj σ σ') (a σ' → e τ)

   e ::= λx.e | e x | op e₁ ... eₙ

cf.       | case x of {{(ϕᵢ → xᵢ) → eᵢ}ᵢ
           | let () = x in e | let (x₁, x₂) = x in e
Pretty-Printing Interpretation

newtype Identity a = Identity a

instance FliPprE Identity Identity where
    ... {- straightforward definition -} ...

pprMode ::
    (∀ a e. FliPprE a e ⇒ e (σ → Doc)) ⇒
    σ → Doc
instance FliPprE GArg GExp where
... {- ??? -} ...

\[ [\Gamma \vdash e : \text{Doc}] :: \text{Grammar} \ [\Gamma] \]

Use umembedding [Atkey+09] to handle \( \Gamma \)

parsingMode ::
(\( \forall a \ e \). FliPprE a e \Rightarrow e (\sigma \rightarrow \text{Doc}) \) \rightarrow 
Grammar \( \sigma \)

(see our paper for detail)
Agenda

- Background: FliPpr
- Embedding FliPpr by Unembedding
  - Interoperable FliPpr
  - Handling Recursions
Motivation

- Explicit treatment of recursions
  - for various parsing algorithms
    - LR(k)
    - Earley
    - ...
  - NB: FliPpr can generate left recursions
    - (Usual) parser combinators and Haskell-level recursions do not terminate
Requirements

- Mutual recursions are necessary
  - \( \text{fix} :: (e \tau \rightarrow e \tau) \rightarrow e \tau \) is a non solution
    - NB: Bekič lemma is not effective
      - unrolling sharings

- Haskell's language support
  - we want to use "recursive" definitions
Our Approach

- Marking for explicit laziness
  - inspired by [Frost+08], [Fischer+11] and
  - the Earley package in Haskell

```haskell
class (FliPprE a e, MonadFix m) => FliPprD m a e where
  mark :: e τ -> m (e τ)

... do wh <- mark $ text " " <*> text "\n" <*> ...
  rec nil <- mark $ text "" <*> space -- 0+ spaces
  space <- mark $ wh <*> nil          -- 1+ spaces

  rec pprT <- mark $ abs $ \x -> ... pprT `app` x ...
  pprF <- mark $ abs $ \x -> ... pprT `app` x ...
```
Derived Combinators

- Pattern-like combinators

```haskell
case_ ... 
  [ unOne $ ..., 
    unSub $ \x y -> ... ]
```

- Better "definitions"

```haskell
rec pprT <- mark $ abs $ \x -> ... pprT `app` x ... 
pprF <- mark $ abs $ \x -> ... pprT `app` x ... 
rec ppr <- defines [True, False] $ \b x -> 
  ... ppr True x ...
```

```haskell
unSub :: FlIPprE a e => (a Exp -> a Exp -> e τ) -> Br a e Exp τ
```
Example

Integration with Haskell's types

data Exp = One | Sub Exp Exp

pprMain :: FliPprD m a e => m (a Exp -> e Doc)
pprMain = do
  rec ppr <- defines [True, False] $ \b x ->
  manyParens $ \
  case_ x
    [ unOne $ text "1",
      unSub $ \x y ->
        group $ parensIf b $
        ppr False x <>
        nest 2
        (line <>
        text "-" <> space <> ppr True y)
    ]
  return $ ppr False

cf. Original

[M&W13]

ppr b x = manyParens (aux b x)
aux _ One       = text "1"
aux b (Sub x y) = group (parensIf b ( 
  ppr False x <>
  nest 2
  (line <>
  text "-" <> space <> ppr True y)))
Our paper also includes:

- `local :: FliPprD m a e ⇒ m (e τ) → e τ`
  - Inverse of "mark", corresponding to `let(rec)`

```
manyParens :: FliPprD m a e ⇒ e Doc → e Doc
manyParens d = local $ do
                rec p <- mark $ d <$> parens p
                return p
```

- Implementation issues
  - Use `unsafeCoarse` for efficiency

- Examples
Related Work

- Embedded invertible/bidirectional languages
  - Inv [Mu+04]
  - Invertible syntax [Rendel&Osterman16]
  - lens variants [Pacheco+10, Kmett]

_all are combinator based (i.e. no binders)_
Related Work

- Applicative lenses [M&W15]
  - conversions from lenses to functions

\[
\text{lift} :: \text{Lens } s \; t \rightarrow (\forall u. \text{Lens } u \; s \rightarrow \text{Lens } u \; t)
\]

\[
\text{unlift} :: (\forall u. \text{Lens } u \; s \rightarrow \text{Lens } u \; t) \rightarrow \text{Lens } s \; t
\]

- with law guarantee
- by Yoneda lemma

- not scalable to guest's binders
  - addressed in HOBiT [M&W18], which is a standalone language
Related Work

- Other pretty-printing combinators
  - [Hughes 95]
  - [Bernardy 17]

(Theoretically) the Original FliPpr can handle them [M&W18b]
Conclusion

- Embedding invertible languages with binders by unembedding [Atkey+09]
  - Case study of FliPpr
    - greater interoperability with Haskell
    - scales to explicit recursions
      - but with a conjecture for mark/local

See our paper for detail. Proof is left for future work.