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 -> 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,…]

$$[\![a \to b]\!] = ([\![a]\!] \to [\![b]\!], [\![b]\!] \to [\![a]\!])$$

- Bidirectional [Foster+05, 07…]

$$\begin{aligned}[\![a \to b]\!] &= \mathit{Lens}\ [\![a]\!]\ [\![b]\!] \\ &= ([\![a]\!] \to [\![b]\!], [\![a]\!] \to [\![b]\!] \to [\![a]\!])\end{aligned}$$

(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])

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

$\lambda x . e$

- Implement inverse semantics

$$Inv[\![\Gamma \vdash e : \tau]\!] :: [\![\tau]\!] \to [\![\Gamma]\!]$$

cf. $[\![\Gamma \vdash e : \tau]\!] :: [\![\Gamma]\!] \to [\![\tau]\!]$

Issues

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

```
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

$$Inv[\![\Gamma \vdash e : \tau]\!] :: [\![\tau]\!] \to [\![\Gamma]\!]$$

cf. $[\![\Gamma \vdash e : \tau]\!] :: [\![\Gamma]\!] \to [\![\tau]\!]$

Approach

❖ Use ***unembedding*** [Atkey+09]

```
unembed :: (∀e. Lam e => e τ) -> DLam () τ
```

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

data DLam env a where
  Var :: In σ env -> DLam env σ
  Abs :: DLam (env,σ) τ -> DLam env (σ -> τ)
  App :: DLam env (σ -> τ) -> DLam env σ -> DLam env τ
```

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

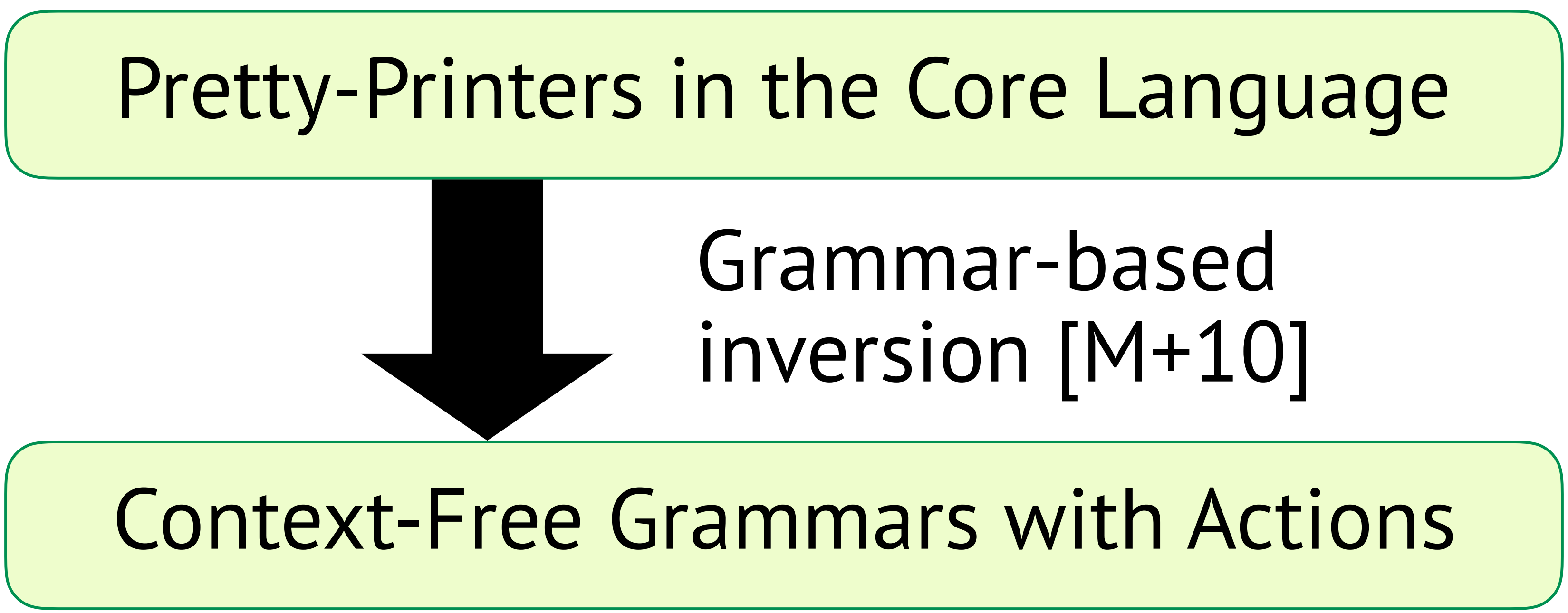
Contributions

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

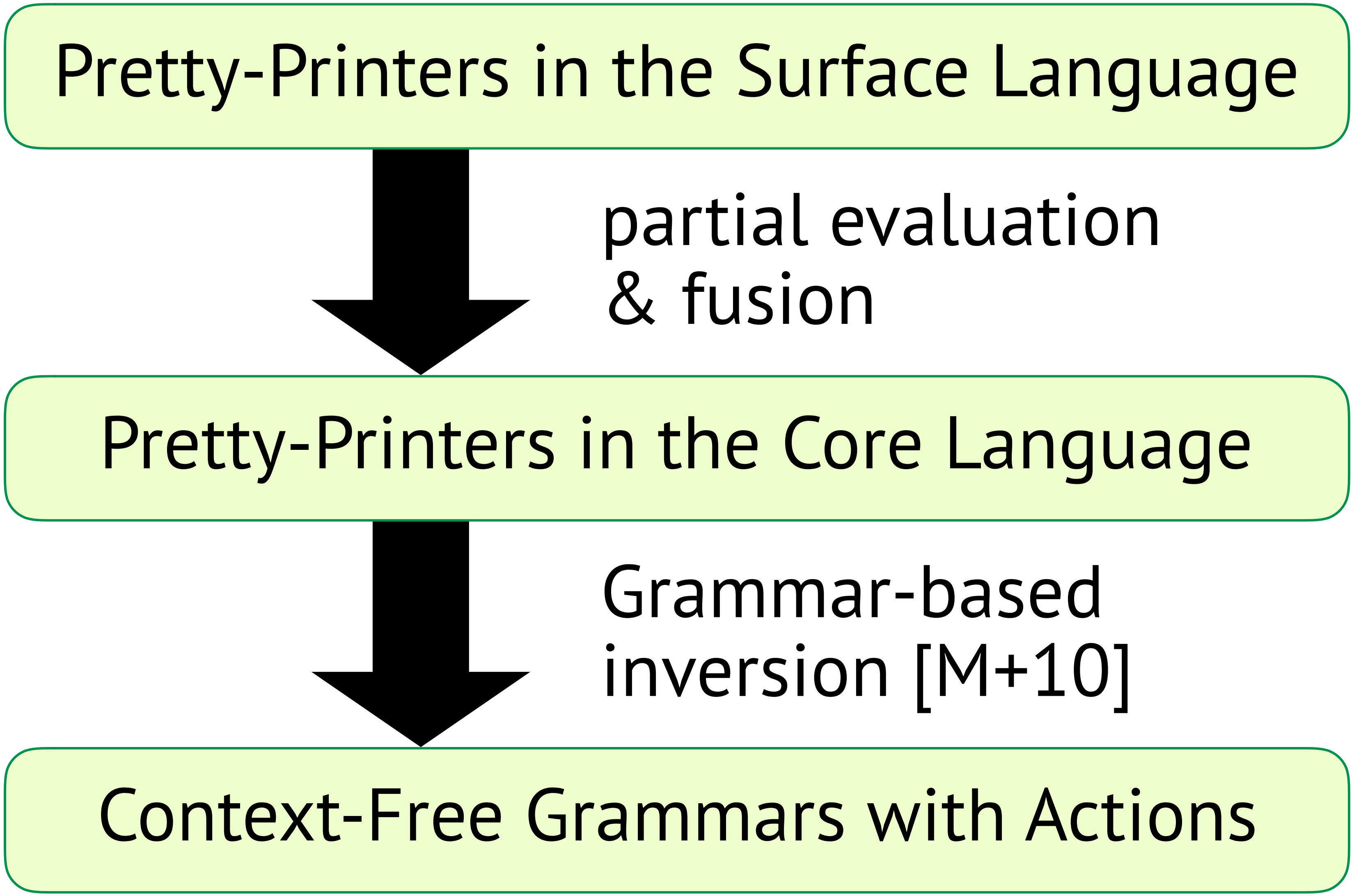
Agenda

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

FliPpr System [M&W13]



FliPpr System [M&W13]



FliPpr Core

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

$$
\begin{aligned}
prog &::= r_1 \ \ldots \ r_n \\
r &::= f(p_1, \ldots, p_n) = e \\
e &::= op\, e_1 \ \ldots \ e_n \\
&\ \ \mid \ \ f \ x_1 \ \ldots \ x_n
\end{aligned}
$$

Arguments must be variables (treeless)

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

FliPpr Surface

partial eval.

fusion

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

```
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

```
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
```

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

…

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)

$$e ::= \lambda x.e \mid e\, x \mid op\, e_1 \ldots e_n$$
$$\mid \ \ \textbf{case}\ x\ \textbf{of}\ \{(\phi_i \to x_i) \to e_i\}_i$$
$$\mid \ \ \textbf{let}\ () = x\ \textbf{in}\ e \mid \textbf{let}\ (x_1, x_2) = x\ \textbf{in}\ e$$

for pattern matching

op : Wadler's combinators

ϕ_i : Haskell-level partial injections

```
type PInj s t = (s -> Maybe t, t -> 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
  ...
data Br a e σ τ = ∀σ'. Br (PInj σ σ') (a σ' -> e τ)
```

pretty-printing result

cf.

$$\begin{aligned} e ::= {} & \lambda x.e \mid e\,x \mid op\,e_1\,\ldots\,e_n \\ \mid {} & \textbf{case}\ x\ \textbf{of}\ \{(\phi_i \to x_i) \to e_i\}_i \\ \mid {} & \textbf{let}\ () = x\ \textbf{in}\ e \mid \textbf{let}\ (x_1, x_2) = x\ \textbf{in}\ e \end{aligned}$$

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
```

Parsing Interpretation

```
instance FliPprE GArg GExp where
  ... {- ??? -} ...
```

$$[\![\Gamma \vdash e : Doc]\!] :: Grammar\ [\![\Gamma]\!]$$

Use umembedding [Atkey+09] to handle Γ

```
parsingMode ::
  (∀ a e. FliPprE a e => e (σ -> Doc)) ->
  Grammar σ
```

(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
  - fix :: (e τ -> e τ) -> e τ 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

```
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

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

```
unSub :: FliPprE a e =>
(a Exp -> a Exp -> e τ) ->
Br a e Exp τ
```

- Better "definitions"

```
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 ...
```

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
```

guest's binders by λ

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

```
lift :: Lens s t -> (∀u. Lens u s -> Lens u t)
unlift :: (∀u. Lens u s -> Lens u t) -> 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.