Mutually Recursive Definition Builders

Kazutaka Matsuda[1] [0000−0002−9747−4899]

Tohoku University, Sendai, Japan kztk@tohoku.ac.jp

Abstract. Mutually recursive definitions are a fundamental concept in programming. This also applies to embedded domain-specific languages (EDSLs), especially when intensional analysis or explicit manipulation of recursive definitions is required. A common approach to model mutually recursive definitions is recursive **let** (**letrec**). However, in the typed setting, **letrec** prevents us from constructing recursive definitions one by one, which is inconvenient when generating EDSL expressions. The difficulty arises because we need to specify the types of recursively bound variables in advance to use **letrec**. In this paper, we propose language constructs that we call *mutually recursive definition builders*, which enable local and one-by-one construction of mutually recursive definitions in a type- and scope-safe manner. We give their formal syntax and typing rules, show an EDSL implementation using higher-order abstract syntax, and discuss their interconvertibility with **letrec**. Additionally, we describe applications of the proposed constructs in an embedded version of FliPpr, an invertible pretty-printing system that involves grammar processing, to highlight their utility in EDSL program generation.

Keywords: EDSL · Mutually Recursive Definitions · Higher-Order Abstract Syntax

1 Introduction

A domain-specific language (DSL) is a programming language designed for a specific problem domain, which often provides tailored abstractions that come with more concise syntax and better efficiency than those of a general-purpose language. An embedded domain-specific language (EDSL), a DSL implemented as a library in a host language, provides extra convenience for both DSL programmers and implementors, as they can access the host language's ecosystem, such as a parser, a type system, editor support, an efficient compiler, and interoperability.

Recursive definition is a fundamental concept in programming. This applies to EDSLs as well. Although explicit handling is often unnecessary in many applications where using the host's recursive definitions is sufficient, it is useful for program optimizations that involve intensional analysis of programs, and is important for grammar manipulations, including grammar transformations and parser generation. Combinators that represent recursive definitions in an EDSL are sometimes called *observable recursions* [13, 37].

For defining a single recursion, the standard fixed-point combinator suffices. When we implement it as an EDSL primitive, if we use higher-order abstract

syntax (HOAS) [12,20,34,35] with an expression type Exp,[1] its type is given as $fix :: (Exp\ a \to Exp\ a) \to Exp\ a$.[2] But, what can we do about *mutual recursion*? Theoretically, fix suffices for mutual recursion via Bekić's lemma [4].

$$fix_2 :: ((Exp\ a, Exp\ b) \to (Exp\ a, Exp\ b)) \to (Exp\ a, Exp\ b)$$
$$fix_2\ f = (fix\ \$\ \lambda x \to fst\ \$\ f\ (x, fix\ \$\ \lambda y \to snd\ \$\ f\ (x, y)), \dots)$$

However, the recursive definitions realized in this way are not useful due to code-size blow-up. Since we cannot share computations among components of $fix_n\ f$, we need $O(n!)$ copies of recursive definitions for n mutual definitions; each ith component of $fix_n\ f$ uses one fix and all the $(n-1)$ components of $fix_{n-1}\ f_i$, where f_i is f with its ith argument set to the variable bound by the fix.

Thus, a practical EDSL needs an operator for mutually recursive definitions when their explicit treatment is required. To mirror the syntax of the recursive **let**, **letrec** $\overline{x = e}$ **in** e', a HOAS representation of the construct looks like:[3]

$$letrec :: Env\ Proxy\ \Delta \to (Env\ Exp\ \Delta \to Env\ Exp\ \Delta) \to (Env\ Exp\ \Delta \to Exp\ t) \to Exp\ t$$

Here, $Env\ f\ \Delta$ is a heterogeneous list [23] of expressions of types Δ, namely, $Env\ f\ [a_1, \dots, a_n] \simeq (f\ a_1, \dots, f\ a_n)$, and $Env\ Proxy\ \Delta$ is a singleton type [14] for Δ, mimicking a dependent function indexed by Δ, realized by reusing Env and a phantom type $Proxy$ defined by **data** $Proxy\ a = Proxy$. Thus, $letrec$ bundles:

$$letrec_0 :: (() \to ()) \to (() \to Exp\ t) \to Exp\ t$$
$$letrec_1 :: (Exp\ a \to Exp\ a) \to (Exp\ a \to Exp\ t) \to Exp\ t$$
$$letrec_2 :: ((Exp\ a, Exp\ b) \to (Exp\ a, Exp\ b)) \to ((Exp\ a, Exp\ b) \to Exp\ t) \to Exp\ t$$
$$\dots$$

However, this straightforward representation and existing variants [3,6,13,22,37] share a common issue: Δ, the types of **letrec**-bound variables, must be known in advance. This global construction reduces modularity. For example, if we want to add a function that will be defined along with existing mutually recursive functions, we need to change not only the function itself but also the rest of the functions, because Δ changes. Also, the exposure of Δ makes some programming difficult, especially when Δ is determined only dynamically. For example, in grammar transformations where grammars are expressed as EDSL programs, we want to generate nonterminals in a resulting grammar on demand, even though there is a known upper bound. Since it is impossible to know how

[1] For presentation simplicity, in informal explanations, we often present types of combinators in the plain HOAS, although the plain HOAS has an issue with interpretations [15] due to exotic terms [11]. HOAS representations without the issue are known [9,11]. If we were to use such representations, fix would be a constructor $Fix :: (v\ a \to Exp\ v\ a) \to Exp\ v\ a$ (in the parametric HOAS [11]) or a typeclass method $fix :: Exp\ f \Rightarrow (f\ a \to f\ a) \to f\ a$ (in the tagless final style [9]).

[2] We use Haskell to explain our ideas throughout this paper.

[3] We abuse the notation to use Δ as a type variable here, while type variables in Haskell must start with lower-case letters.

many nonterminals will be generated before a transformation, it is difficult or at least nontrivial to realize the transformation via *letrec*-like combinators. Thus, we aim for a more fine-grained construction of mutually recursive definitions, enabling one to work with a single binding at a time without referring to Δ.

In this paper, we propose *mutually recursive definition builders* (MRD builders, for short) to construct mutually recursive definitions from single bindings without knowing the whole bindings (the Δ above). More specifically, MRD builders consist of four constructs **freeze**, **push**, **letr**, and **ret** that manage a worklist of expressions to be named and put in the **letrec** bindings. Intuitively, **ret** e starts building with an expression **letrec in** e and the empty worklist, **push** e d pushes an expression e into the worklist of a builder d, **letr** $x.\, d$ pops the head expression from the worklist of a builder d and names it x, and **freeze** d finishes building. For example, with them, a monolithic expression **let rec** $x_1 = e_1, \ldots, x_n = e_n$ **in** e' is broken down into the compound form

$$\textbf{freeze}\ \big(\textbf{letr}\ x_1.\ \ldots \textbf{letr}\ x_n.\, \textbf{push}\ e_n\ (\ldots(\textbf{push}\ e_1\ (\textbf{ret}\ e'))\ldots)\big)$$

which pushes expressions $e_1, \ldots, e_n$ to the worklist in this order and then pops them with names $x_n, \ldots, x_1$ to produce the bindings $x_1 = e_1, \ldots, x_n = e_n$.

Even more interesting is our EDSL representation of the proposed constructs. Leveraging the power of tagless final representation [9], MRD builders can be implemented as an "extension kit" for an existing non-recursive EDSL. Specifically, our approach works for any EDSL whose expression type constructor *Exp* has kind $k \to \mathit{Type}$ for arbitrary k, which covers both parametric HOAS [11] and the tagless final style [9]. This design ensures that we can provide our core and derived combinators as a library, rather than an implementation technique that needs to be applied to EDSLs on a one-by-one basis. Among such combinators, the most interesting one is *letr1* :: $\mathit{Defs}\ f \Rightarrow (f\ a \to \mathit{DefM}\ f\ (f\ a, r)) \to \mathit{DefM}\ f\ r$, which intuitively bundles **push** and **letr**. With *letr1*, we can compositionally construct combinators for mutually recursive definitions of an arbitrary number of bindings (Section 2.3), as well as (a tupled [10,19] form of) the *letrec* combinator (Section 3.1). We can even handle the case where the number of bindings is determined dynamically (*mkDivisibleBy* in Section 2.3).

In summary, our contributions are:

- We propose combinators for mutually recursive definitions that enable the "local" construction of each object to be defined mutually recursively (Section 2.1). We also provide several derived combinators to make programming easier (Sections 2.2, 2.3 and 2.5), and show semantics examples (Section 2.4).
- We discuss the relationship between the proposed constructs and **letrec** (Section 3). As a standalone DSL syntax, they are straightforwardly interconvertible. As an EDSL syntax, the conversion from **letrec** (i.e., **letrec** on top of MRD builders) remains easy (Section 3.1), but the opposite (MRD builders on top of **letrec**) requires additional effort [1,2,27] due to the implicit nature of the whole bindings (Section 3.2).
- We discuss the theoretical background of the proposed combinators (Section 4). Their design is inspired by the trace operator in category theory [21].

Syntax

$$
\begin{array}{lll}
e & ::= \cdots \mid \mathbf{freeze}\ d & \text{(DSL expressions)} \\
d & ::= \mathbf{ret}\ e \mid \mathbf{push}\ e\ d \mid \mathbf{letr}\ x.d & \text{(builders)} \\
\sigma, \tau & ::= \ldots & \text{(DSL types)} \\
A & ::= \tau_1, \ldots, \tau_n \rhd \tau & \text{(builder types)}
\end{array}
$$

(Additional) typing rules: $\boxed{\Gamma \vdash e : \tau}$ **and** $\boxed{\Gamma \vdash d : \overline{\sigma} \rhd \tau}$

$$
\frac{\Gamma \vdash d : \rhd\tau}{\Gamma \vdash \mathbf{freeze}\ d : \tau} \quad
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ret}\ e : \rhd\tau} \quad
\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash d : \overline{\sigma} \rhd \tau}{\Gamma \vdash \mathbf{push}\ e\ d : \sigma, \overline{\sigma} \rhd \tau} \quad
\frac{\Gamma, x : \sigma \vdash d : \sigma, \overline{\sigma} \rhd \tau}{\Gamma \vdash \mathbf{letr}\ x.d : \overline{\sigma} \rhd \tau}
$$

Fig. 1: Proposed constructs and their typing rules (as a standalone DSL)

– We demonstrate nontrivial use cases of MRD builders in the embedded version [30] of FliPpr [29,31], a language for invertible pretty-printers (Section 5).

An implementation of MRD builders is provided as a part of the embedded FliPpr, whose repository is hosted at https://github.com/kztk-m/flippre.

2 Our Proposal: Mutually Recursive Definition Builders

In this section, we propose MRD builders. We first show their core constructs (Section 2.1) and then discuss a derived interface (Section 2.2).

2.1 Core Constructs

Although our focus is on embedded DSLs, we first illustrate our proposed constructs in the context of a standalone DSL, as presented in Fig. 1. As mentioned in Section 1, MRD builders consist of four constructs: **freeze**, **push**, **letr**,[4] and **ret**. They come with an additional syntactic category d, called a *builder*. A builder d of type $\tau_1, \ldots, \tau_n \rhd \tau$ represents a pair of a partially constructed **letrec** of type τ and a worklist (more precisely, a stack) of expressions $e_1, \ldots, e_n$ to be named and put in the **letrec** bindings, where each e_i has type τ_i. The operator **ret** e produces a **letrec** with the empty bindings, i.e., **letrec in** e, together with the empty worklist; **push** $e\ d$ pushes e into the worklist part of d; **letr** $x.d$ pops the head e of the worklist in d, names it x, and adds $x = e$ to the binding part; and **freeze** d finally converts it to the standard **letrec**, provided that the worklist is empty. The crucial point is that the types of already constructed bindings are hidden in the typing rules, while the worklist expressions can refer to those hidden bindings. These fine-grained constructs for mutually recursive definitions have no clear benefits in a standalone DSL. Instead, their usefulness becomes evident in EDSLs, where we programmatically construct EDSL expressions.

The above intuition (the worklist and partially constructed **letrec**) for the MRD builders suggests the following identity, provided $x \notin \mathrm{fv}(e)$.

$$\mathbf{push}\ e\ (\mathbf{letr}\ x.\, \mathbf{push}\ e_x\ d) \equiv \mathbf{letr}\ x.\, \mathbf{push}\ e_x\ (\mathbf{push}\ e\ d) \qquad (\textsc{PushSliding})$$

[4] The name of **letr** is a portmanteau of **let** and the **tr**ace operator [21] in category theory (Section 4). It also suggests partially constructed **letrec**s.

Hence, when some definition bodies do not refer to some bound variables, there is more than one way to construct the mutually recursive definitions. We could avoid redundancy in representation by removing **push** and making **ret** take expressions to be pushed, as **ret** $e_1, \ldots, e_n\ e$. We, however, keep the redundancy, as removing it by having a more "core" syntax would substantially complicate the derived combinators that we introduce later.

If we use the tagless final [9] representation of HOAS [12, 20, 34, 35], these constructs can be expressed as the following methods in a type class *Defs*.[5]

```
class Defs (f :: k → Type) where
  -- D f [τ1, ..., τn] τ represents the builder type τ1, ..., τn ▷ τ
  data D f :: [k] → k → Type
  retD ::    f τ → D f '[] τ
  freezeD :: D f '[] τ → f τ
  pushD ::   f α → D f αs τ → D f (α : αs) τ
  letrD ::   (f α → D f (α : αs) τ) → D f αs τ
```

Thanks to the extensibility of the tagless final style, the definition of the type class *Defs f* is given independently of any concrete EDSL expression type $f\ \alpha$. By using them, we can construct mutually recursive definitions **letrec** $x_1 = e_1, \ldots, x_n = e_n$ **in** e as

$$\mathit{freezeD}\ \$\ \mathit{letrD}\ \$\ \lambda x_1 \to \cdots \to \mathit{letrD}\ \$\ \lambda x_n \to \mathit{pushD}\ e_n\ \$\ \ldots\ \$\ \mathit{pushD}\ e_1\ \$\ \mathit{retD}\ e$$

while no single construct refers to the whole binding information.

Having appropriate instances of *Defs* makes our combinators (Section 2.2) accessible to EDSLs in other representations. It is straightforward to have a *Defs* instance for an EDSL in the parametric HOAS [11]. We can even use de Bruijn-indexed terms through Atkey's unembedding [1, 2, 27].

2.2 Monadic Surface Interface

We can program directly with the proposed constructs as above, but programming becomes much easier with the following *derived* interface.

```
letr1 :: Defs f ⇒ (f α → DefM f (f α, r)) → DefM f r
local :: Defs f ⇒ DefM f (f α) → f α
```

Here, *DefM f* is a monad defined on top of our primitives, equipped with the two operations above. We shall postpone their definitions until Section 2.5.

[5] This code assumes GHC extensions `PolyKinds`, `TypeFamilies`, `DataKinds`, and `TypeOperators`: `PolyKinds` makes f polymorphic in its DSL type domain k, `TypeFamilies` allows the associated datatype D, and `DataKinds` and `TypeOperators` enable us to use (:) and [] in the type level, where the latter needs to be prefixed by a tick "'" for the disambiguation from the list type constructor (in Haskell, [] *Int* and [*Int*] are synonyms).

Intuitively, this interface conceals $D\ f\ \alpha s\ \tau$ objects under nested pairs in the monad $DefM\ f$. More specifically, $D\ f\ [\alpha_1, \ldots, \alpha_n]\ \tau$ in the core interface corresponds to $DefM\ f\ (f\ \alpha_1, (f\ \alpha_2, (\ldots, (f\ \alpha_n, f\ \tau)\ldots)))$ in this interface. An immediate consequence is that we can use standard functions that manipulate pairs and monads. Moreover, as we will demonstrate shortly, the monadic interface provides additional modularity and programming flexibility (Section 2.3).

2.3 Programming with the Monadic Interface

To demonstrate the utility of the combinators, we use a toy language below.

```
class ToyL (f :: Type → Type) where
  int :: Int → f Int;                  bool :: Bool → f Bool
  (+.) :: f Int → f Int → f Int;       (−.) :: f Int → f Int → f Int
  if_ :: f Bool → f a → f a → f a;     eq :: f Int → f Int → f Bool
  lam :: (f a → f b) → f (a → b);      app :: f (a → b) → f a → f b
```

For simplicity, we assume that DSL types have kind *Type*, the kind for Haskell types, though we can use any datatype promoted by `DataKinds`.

Basics of the Monadic Interface Let us consider defining *even* mutually recursively with *odd*. For comparison, using the core combinators, following the intuition described in Section 2.1, this function can be implemented as follows.

```
even1 :: (ToyL f, Defs f) ⇒ f (Int → Bool)
even1 = freezeD $ letrD $ λeven → letrD $ λodd →
  pushD (lam $ λn → if_ (n `eq` int 0) (bool False) (app even $ n −. int 1)) $
  pushD (lam $ λn → if_ (n `eq` int 0) (bool True) (app odd $ n −. int 1)) $
  retD even
```

Observe that the body of *odd* comes before that of *even*, as MRD builders perform stack-based construction of mutually recursive definitions.

Following the correspondence above, we can derive equivalent code with the monadic interface systematically from $even_1$ above, by replacing *freezeD* with *local*, *pushD* e with *fmap* $(\lambda x \to (e, x))$, *letrD* with *letr1*, and *retD* with *pure* (a synonym of *return*).

```
even2 :: (ToyL f, Defs f) ⇒ f (Int → Bool)
even2 = local $ letr1 $ λeven → letr1 $ λodd →
  let e_body = lam $ λn → if_ (n `eq` int 0) (bool True) (app odd $ n −. int 1)
      o_body = lam $ λn → if_ (n `eq` int 0) (bool False) (app even $ n −. int 1)
  in fmap (λx → (o_body, x)) $ fmap (λx → (e_body, x)) $ pure even
```

(The last line can be simplified to $pure\ (o_{\mathrm{body}}, (e_{\mathrm{body}}, even))$, but we keep the redundant form for comparison.)

At this point, this monadic interface appears to have no significant difference from the core version, but it actually provides additional modularity. For example,

with the monadic interface, we can return both *even* and *odd* at the same time, leveraging the fact that *pure* can take an arbitrary Haskell expression, while *retD* can only take an EDSL expression.

```
mkEvenOdd :: (ToyL f, Defs f) ⇒ DefM f (f (Int → Bool), f (Int → Bool))
mkEvenOdd = letr1 $ λeven → letr1 $ λodd →    -- Observe: no local here.
  let e_body = lam $ λn → if_ (n `eq` int 0) (bool True) (app odd $ n −. int 1)
      o_body = lam $ λn → if_ (n `eq` int 0) (bool False) (app even $ n −. int 1)
  in fmap (λx → (o_body, x)) $ fmap (λx → (e_body, x)) $ pure (even, odd)
```

This function can be used as **do** $\{(even, odd) \leftarrow mkEvenOdd; \ldots even \ldots odd \ldots\}$. A crucial point is that, in this code snippet, *even* and *odd* refer to the defined variables, while $even_1$ and $even_2$ above refer to a whole expression corresponding to **letrec** $even = \ldots; odd = \ldots$ **in** *even*—this whole expression will be copied every time $even_1$ and $even_2$ are used.

Divisibility Test Generator The pattern in *even* can be easily extended to a general divisibility test implemented by a mutually recursive definition **letrec** $f_0 = e_0, \ldots, f_{m-1} = e_{m-1}$ **in** f_0, where, if the given argument is 0, e_i returns true if $i = 0$ and false otherwise, and if the argument is not 0, it invokes $f_{(i-1) \bmod m}$ for the predecessor of the argument.

We can define a divisibility test generator with *letr1* based on the following idea. By recursion on n, we gather names generated by *letr1*, so that we can use these names (which correspond to $f_0, \ldots, f_{n-1}$ above) in their corresponding bodies that we produce at the base case. The induction step picks an appropriate definition. This idea, with a slight generalization that we index names and bodies as $\{f_{k_i}\}_i$ and $\{e_{k_i}\}_i$ for a given list $k_1, \ldots, k_n$, can be implemented as the following function *letr1s*.

```
letr1s :: (Defs f, Eq k) ⇒
          [k] → ((k → f α) → DefM f (k → f α, r)) → DefM f r
letr1s ks0 h = fmap snd $ go ks0 (const $ error "out of bounds")
  where
    go [] names = h names
    go (k : ks) names = letr1 $ λf_k → do
      (bodies, r) ← go ks (λx → if x == k then f_k else names x)
      let e_k = bodies k in pure (e_k, (bodies, r))
```

Given *letr1s*, the definition of *mkDivisibleBy* is straightforward.

```
mkDivisibleBy :: (ToyL f, Defs f) ⇒ Int → DefM f (f (Int → Bool))
mkDivisibleBy m = letr1s [0 .. m − 1] $ λself → do
  let h k = lam $ λn → if_ (n `eq` int 0)
                           (bool $ k == 0)
                           (app (self ((k − 1) `mod` m)) $ n −. int 1)
  pure (h, self 0)
```

Defining *mkDivisibleBy* with *letrec* would be nontrivial, because the types of recursively defined variables, which are static, depend on a given integer, which is dynamic. The construction directly with the core combinators requires much effort due to *go* in *letr1s* returning $k \to f\ \alpha$. Recall that d of *letrD* $\$\ \lambda x \to d$ and the whole expression must be of builder types. Continuation-passing style addresses this issue [8,38]; *DefM* follows the idea (Section 2.5).

More Derived Combinators Let us go back to *mkEvenOdd*, which is still awkward compared with ordinary recursive definitions. Ideally, we would like to define DSL-level recursive definitions in syntax similar to Haskell-level recursive definitions. We can mimic this by leveraging GHC extensions `RecursiveDo` with `QualifiedDo` or `RebindableSyntax`, which allow us to replace *mfix* used to desugar recursive **do** notation with our own version, which may have a different type from the standard $mfix :: MonadFix\ m \Rightarrow (a \to m\ a) \to m\ a$.

To define our version of *mfix*, we prepare the following type class, which provides the method *letr* to generalize *letr1* and bind multiple definitions at once.

```
class Monad m ⇒ RecArg m t where
  letr :: (t → m (t, r)) → m r
```

Then, we define several instances.

```
newtype W a = W {unW :: a}   -- a newtype wrapper
instance Defs f ⇒ RecArg (DefM f) (W (f a)) where
  letr h = letr1 (fmap (first unW) ∘ h ∘ W)
instance (RecArg m a, RecArg m b) ⇒ RecArg m (a, b) where
  letr h = letr $ λa → letr $ λb → do
    ((a', b'), r) ← h (a, b)
    pure (b', (a', r))
instance (RecArg m a, RecArg m b, RecArg m c)
         ⇒ RecArg m (a, b, c) where
  letr h = letr $ λa → letr $ λb → letr $ λc → do
    ((a', b', c'), r) ← h (a, b, c)
    pure (c', (b', (a', r)))
```

We can go further for any type isomorphic to a product of $f\ a_1, \ldots f\ a_n$, including tuples of size four or more, as well as bounded-domain functions such as $Bool \to a$ (provided that $RecArg\ m\ a$ holds). Above, we use *first* from *Control.Arrow*, which here is instantiated as $first :: (a \to b) \to (a, x) \to (b, x)$. A subtlety is that, if we were to declare an instance of $RecArg\ (DefM\ f)\ (f\ a)$, it would overlap with other instance declarations (e.g., f can match $(,)\ a$ or $(,)\ a\ b$) and confuse Haskell instance resolution, reducing usability. Thus, for the tagless final style, we need to use a newtype wrapper W to indicate the base cases (leaves of the product).

By using *letr*, we can define our version of *mfix*.

```
mfixMRD :: RecArg m t ⇒ (t → m t) → m t
mfixMRD h = letr $ λt → do {t' ← h t; pure (t', t)}
```

By exporting *mfixMRD* as *mfix* in a module, and importing the module under, say, the name F in another module, we can repurpose recursive **do** syntax.

```
-- mkEvenOdd2 has the same type as mkEvenOdd.
mkEvenOdd2 = F.do  -- QualifiedDo
  -- To desugar do, F also re-exports the standard return, (>>=), and (fail).
  rec even <- pure $ W $ lam $ \n ->
                if_ (n `eq` int 0) (bool True) (app (unW odd) $ n - int 1)
       odd <-  pure $ W $ lam $ \n ->
                if_ (n `eq` int 0) (bool False) (app (unW even) $ n - int 1)
  pure (unW even, unW odd)
```

Notice that the required use of W/unW stems from a bare type variable f in the tagless final representation. Hence, if our EDSL expression type is guarded by a type constructor, e.g., when we use wrapper types, or use parametric HOAS (PHOAS) [11] representation, we can remove W and even *lam*/*app* in the above definition by giving appropriate instances of *RecArg*.

Derived letrec. Following a similar idea to *letr* definitions, we can even derive (a tupled [10,19] version of) *letrec* given in Section 1 on top of our combinators; see Section 3.1 for the concrete construction. This means, in programming, we can express anything expressible with *letrec*, the main alternative.

Issue with mfix-like combinators on top of letrec. Unlike *mkDivisibleBy* in Section 2.3, *mfixMRD* assumes that the types of recursive bindings are statically determined to pick an appropriate *RecArg* instance. This suggests that we can have a similar *mfix*-like combinator even with *letrec*. This is possible, but impractical as it copies bindings. To explain why, let us focus on the two-variable case, $letrec_2$, in Section 1. Using a continuation monad $C\ f\ a = \forall \tau.\, (a \to f\ \tau) \to f\ \tau$, we can write its type as $letrec_2 :: ((f\ a, f\ b) \to (f\ a, f\ b)) \to C\ f\ (f\ a, f\ b)$. This version is closer to what we want $mfixLetRec_2 :: ((f\ a, f\ b) \to C\ f\ (f\ a, f\ b)) \to C\ f\ (f\ a, f\ b)$, where the argument function is also monadic. We can convert $m :: C\ f\ (f\ a, f\ b)$ to $(f\ a, f\ b)$ by $(m\ fst, m\ snd)$, but it copies a context surrounding the continuation in m. We can implement a general *mfixLetRec* in this way, but the resulting combinator is not useful as nesting copies the context, which consists of mutually recursive bindings. To make the construction feasible, we need to know that such a context consists only of local bindings, which are open to extension. These are builders (d in Fig. 1). Our *DefM* is defined as a continuation monad for *builders*, instead of for expressions (Section 2.5).

2.4 Defining Semantics

We now discuss implementing the semantics of such EDSLs. Let us consider giving a semantics for the simple language, which is achieved by giving a certain semantic domain S that is both *ToyL S* and *Defs S*.

To define the standard semantics, we choose $S = Identity$, where *Identity* is the identity functor defined in *Data.Functor.Identity* as **newtype** *Identity a* =

Identity $\{runIdentity :: a\}$. We focus on *Defs Identity* here; giving an instance of *ToyL* for *Identity* is straightforward, and is hence omitted. In the standard semantics, where we implement EDSL-level recursive definitions via Haskell-level recursive definitions, we use nonempty heterogeneous lists with distinguished last elements to implement *D Identity*, and realize *letrD* by knot-tying.

```
instance Defs Identity where
  data D Identity αs τ = DI (Env Identity αs) (Identity τ)   -- init-last pair
  freezeD (DI _ r) = r
  retD r = DI ENil r
  pushD e (DI es r) = DI (ECons e es) r
  letrD h = let DI (ECons a as) r = h a in DI as r   -- relying on laziness
data Env (f :: k → Type) (αs :: [k]) where   -- heterogeneous list
  ENil ::   Env f '[]
  ECons :: f α → Env f αs → Env f (α : αs)
```

Here is an evaluation example.

```
> let isDivisible4 = local $ mkDivisibleBy 4
> [runIdentity (isDivisible4 `app` int i) | i ← [1 .. 10]]
[False, False, False, True, False, False, False, True, False, False]
```

Another example is printing semantics, where we use de Bruijn levels (nesting depths of binders) for variable names.

```
type Level = Int; type Prec = Int
type P = Level → Prec → String
data PSem a = PSem {pr :: P}
```

We could print MRD builders as the syntax given in Fig. 1, but then our printing semantics would be no different from those for other tagless final EDSLs. Our *Defs* class follows the standard tagless final encoding of the syntactic category d (Fig. 1) except for a slight deviation: the associated type D is used to avoid an extra parameter for *Defs*. Thus, we consider printing programs as if **letrec** were used, following the informal meanings of MRD builders.

```
instance Defs PSem where
  newtype D PSem αs τ = DP {unDP :: Level → ([(String, P)], [P], P)}
  retD e = DP $ const ([], [], pr e)
  pushD e (DP p) = DP $ λl → let (bs, es, r) = p l in (bs, pr e : es, r)
  letrD h = DP $ λl →
    let f = "f" <> show l
        (bs, e : es, r) = unDP (h (PSem $ λ_ _ → f)) (l + 1)
    in ((f, e) : bs, es, r)
  freezeD (DP p) = PSem $ λlevel _ →
    let (bs, _, e) = p level
        level' = level + length bs
```

$$\textbf{in}\ \texttt{"letrec\textbackslash n"} \mathbin{<>} \mathit{mconcat}\ [f \mathbin{<>} \texttt{" = "} \mathbin{<>} b\ \mathit{level}'\ 0 \mathbin{<>} \texttt{"\textbackslash n"} \mid (f, b) \leftarrow \mathit{bs}]$$
$$\mathbin{<>} \texttt{"in "} \mathbin{<>} e\ \mathit{level}'\ 0$$

Here, *Level* in *DP* is used to name *letrD*-bound variables; naming *lam*-bound variables is deferred to *freezeD*, which performs the main job of printing. By the instances, we can print EDSL expressions.

```
> putStrLn $ pr (local $ mkDivisibleBy 2) 0 0
letrec
f0 = \x2 -> if x2==0 then True else f1 (x2-1)
f1 = \x2 -> if x2==0 then False else f0 (x2-1)
in f0
```

Not all semantics are straightforward. For example, conversion to **letrec** requires additional effort (Section 3.2). Another example is the nullability check for regular/context-free grammars, which requires iterations. Using *Const Bool* with the constant functor *Const* works, but is inefficient, as *letrD h* may run h twice: first for *Const False*, and then for *Const True* if needed.

2.5 Concrete Definition of *DefM*

We conclude the section by showing the definition of the monad *DefM* and its operations *letr1* and *local*. The definition of *DefM* is analogous to the codensity monad [25, 39].[6]

$$\textbf{newtype}\ \mathit{DefM}\ f\ a = \mathit{DefM}\ \{\mathit{unDefM} :: \forall \alpha s\ \tau.\ (a \to D\ f\ \alpha s\ \tau) \to D\ f\ \alpha s\ \tau\}$$

We omit its straightforward *Functor*, *Applicative*, and *Monad* instance declarations. A similar continuation monad is used in staged programming for binding-time improvement [8, 26, 38]. Such a monad provides programming flexibility, particularly for **let** generation [8, 38], by allowing code generation functions to return non-code types. Then, we give the definitions of *letr1* and *local*.

$$\mathit{letr1} :: \mathit{Defs}\ f \Rightarrow (f\ \alpha \to \mathit{DefM}\ f\ (f\ \alpha, r)) \to \mathit{DefM}\ f\ r$$
$$\mathit{letr1}\ h = \mathit{DefM}\ \$\ \lambda k \to \mathit{letrD}\ \$\ \lambda a \to \mathit{unDefM}\ (h\ a)\ \$\ \lambda(b, r) \to \mathit{pushD}\ b\ (k\ r)$$

$$\mathit{local} :: \mathit{Defs}\ f \Rightarrow \mathit{DefM}\ f\ (f\ \alpha) \to f\ \alpha$$
$$\mathit{local}\ m = \mathit{freezeD}\ \$\ \mathit{unDefM}\ m\ \mathit{retD}$$

The idea behind the implementation of *letr1* is that we use *letrD* to produce an argument for h and convert tuples to $D\ f$-typed values, following the intuitive correspondence (Section 2.2). In particular, the result of h is a pair (b, r). For its second component r, we apply k to obtain $k\ r :: D\ f\ \alpha s\ \tau$, making the pair ready for *pushD*, yielding a value of type $D\ f\ (\alpha : \alpha s)\ \tau$, as required by *letrD*. Note that the answer types of the two continuations (k and $\lambda(b, r) \to \ldots$) above differ, which justifies the use of the codensity-like monad where αs and τ are universally quantified. In contrast, the definition of *local* is straightforward.

[6] We say "analogous" here as we have not confirmed the functoriality of $D\ f$. There are no constructs to transform αs and τ in $D\ f\ \alpha s\ \tau$. However, they conceptually appear positively in d as defined in Fig. 1. Thus, we conjecture potential functoriality.

3 Relationship to letrec

In this section, we discuss interconversion between our proposed constructs and **letrec** $\overline{x = e}$ **in** e'. While the conversion from **letrec** can be performed on HOAS representations (Section 3.1), the opposite direction is performed on the DSL syntax given in Fig. 1 rather than on the HOAS representations of these constructs (Section 3.2). This effectively requires a conversion from HOAS to de Bruijn-indexed terms [1, 2, 27] for the conversion to **letrec**.

3.1 Conversions from letrec: letrec on Top of Our Constructs

The conversion from **letrec** is straightforward, and is given similarly to the definitions of *letr* in Section 2.2. Specifically, we define the conversion by giving a derived combinator *letrec* on top of MRD builders, as below.

```
letrec :: (Defs f) ⇒ Env Proxy αs → (Env f αs → (Env f αs, f τ)) → f τ
letrec sh h = local $ letrecM sh (pure ∘ h)
letrecM :: (Defs f) ⇒ Env Proxy αs
             → (Env f αs → DefM f (Env f αs, r)) → DefM f r
letrecM ENil h = snd <$> h ENil
letrecM (ECons _ sh) h = letr1 $ λx → letrecM sh $ λxs → do
  (vvs, r) ← h (ECons x xs)
  case vvs of    -- trick to tell GHC that this matching is exhaustive.
    ECons v vs → pure (vs, (v, r))
```

This *letrec* differs slightly from the one in Section 1; this version is tupled [10, 19].

The function *letrec* is defined (via *letrecM*) by induction on its first argument *sh*, which intuitively is a value-level representation of the type αs; in other words, *Env Proxy* αs serves as a singleton type [14] for αs. Existing *letrec*-like combinators (e.g., [22]) also require such shape information in different forms. Otherwise, since we have no information about αs, the interpretation of *letrec* is limited; we cannot give an argument to h without using knot-tying (Section 6).

3.2 Conversion to letrec: Our Constructs on Top of letrec

In contrast, the opposite direction requires more effort when *HOAS* is used. To bypass the difficulty, we first show a type-directed translation relation, which follows the intuition given in Section 2.1—$d : \overline{\sigma} \rhd \tau$ represents a pair of a partially constructed **letrec** of type τ and a worklist of expressions of type $\overline{\sigma}$. Then, we discuss why this is not straightforward on the HOAS representation.

Let us write μ for recursive bindings of the form $\overline{x = e}$. Then, we can present the typing rule for μ as follows.

$$\frac{\{\Gamma, \Delta \vdash \mu(x) : \Delta(x)\}_{x \in \mathrm{dom}(\mu)}}{\Gamma, \Delta \vdash \mu : \Delta}$$

Recall that, in Section 2.1, we explained that d intuitively represents a partially constructed **letrec** μ **in** e with a worklist $\overline{e}$. Formally, such pairs serve as translation results of builders d. Thus, we need to give them types to define our type-preserving translation. A caveat is that the worklist expressions can use the **letrec**-bound variables. Emphasizing this dependency, we write these intermediate objects as **letrec** $\mu \,|\, \overline{e}$ **in** e, instead of pairs of **letrec** μ **in** e and $\overline{e}$. Recall that we typed $\Gamma \vdash d : \overline{\sigma} \rhd \tau$ (Section 2.1), and correspondingly, we give a typing rule for these intermediate objects as follows.

$$\frac{\exists \Delta.\quad \Gamma, \Delta \vdash \mu : \Delta \quad \{\Gamma, \Delta \vdash e_i : \sigma_i\}_{1 \le i \le n} \quad \Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \textbf{letrec}\ \mu \,|\, e_1, \ldots, e_n\ \textbf{in}\ e : \sigma_1, \ldots, \sigma_n \rhd \tau}$$

When the worklist is empty ($n = 0$ above), the rule is similar to the standard typing rule of **letrec** in simply-typed systems, witnessing that **letrec** $\mu \,|\,$ **in** e is ready to convert to **letrec** μ **in** e. This is how **freeze** is processed.

Now, we are ready to define our conversion $\Gamma \vdash d \rightsquigarrow$ **letrec** $\mu \,|\, \overline{e}$ **in** $e : \overline{\sigma} \rhd \tau$, which is read as $\Gamma \vdash d : \overline{\sigma} \rhd \tau$ is converted to $\Gamma \vdash$ **letrec** $\mu \,|\, \overline{e}$ **in** $e : \overline{\sigma} \rhd \tau$. This conversion is defined mutually recursively with $\Gamma \vdash e \rightsquigarrow e' : \sigma$ that converts $\Gamma \vdash e : \sigma$ to $\Gamma \vdash e' : \sigma$. These rules are presented in a type-directed style to highlight the treatment of typing environments and their type-preserving nature, whereas the conversions themselves are syntax-directed.

$$\frac{\Gamma \vdash d \rightsquigarrow \textbf{letrec}\ \mu \,|\,\ \textbf{in}\ e : \rhd\tau}{\Gamma \vdash \textbf{freeze}\ d \rightsquigarrow \textbf{letrec}\ \mu\ \textbf{in}\ e : \tau} \qquad \frac{\Gamma \vdash e \rightsquigarrow e' : \sigma}{\Gamma \vdash \textbf{ret}\ e \rightsquigarrow \textbf{letrec}\ \mid\ \textbf{in}\ e' : \rhd\tau}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \sigma \quad \Gamma \vdash d \rightsquigarrow \textbf{letrec}\ \mu \,|\, \overline{e}\ \textbf{in}\ e : \overline{\sigma} \rhd \tau}{\Gamma \vdash \textbf{push}\ e\ d \rightsquigarrow \textbf{letrec}\ \mu \,|\, e', \overline{e}\ \textbf{in}\ e : \sigma, \overline{\sigma} \rhd \tau}$$

$$\frac{\Gamma, x : \sigma \vdash d \rightsquigarrow \textbf{letrec}\ \mu \,|\, e', \overline{e}\ \textbf{in}\ e : \sigma, \overline{\sigma} \rhd \tau}{\Gamma \vdash \textbf{letr}\ x.d \rightsquigarrow \textbf{letrec}\ \mu, x = e' \,|\, \overline{e}\ \textbf{in}\ e : \overline{\sigma} \rhd \tau}$$

Other than **freeze**, the rules for $\Gamma \vdash e \rightsquigarrow e' : \sigma$ are defined so that they process subexpressions recursively. The last rule requires an explanation of why it preserves types. We show that we can conclude $\Gamma \vdash$ **letrec** $\mu, x = e' \,|\, \overline{e}$ **in** $e : \overline{\sigma} \rhd \tau$ from $\Gamma, x : \sigma \vdash$ **letrec** $\mu \,|\, e', \overline{e}$ **in** $e : \sigma, \overline{\sigma} \rhd \tau$. Unfolding the definition of $\Gamma, x : \sigma \vdash$ **letrec** $\mu \,|\, e', \overline{e}$ **in** $e : \sigma, \overline{\sigma} \rhd \tau$ in the premise, we have $\Gamma, x : \sigma, \Delta \vdash \mu : \Delta$, $\Gamma, x : \sigma, \Delta \vdash e' : \sigma$, $\{\Gamma, x : \sigma, \Delta \vdash e_i : \sigma_i\}_{1 \le i \le n}$ for $\overline{e} = e_1, \ldots, e_n$ and $\overline{\sigma} = \sigma_1, \ldots, \sigma_n$, and $\Gamma, x : \sigma, \Delta \vdash e : \tau$ for some Δ. Then, taking $\Delta' = \Delta, x : \sigma$, we have $\Gamma, \Delta' \vdash \mu, x = e' : \Delta'$, $\{\Gamma, \Delta' \vdash e_i : \sigma_i\}_{1 \le i \le n}$, and $\Gamma, \Delta' \vdash e : \tau$, concluding $\Gamma \vdash$ **letrec** $\mu, x = e' \,|\, \overline{e}$ **in** $e : \overline{\sigma} \rhd \tau$ as requested.

This existential nature of Δ makes the conversion with the HOAS representation difficult. To explain this, suppose that Δ were explicitly exposed in the builder types and in the above translation. In this case, the right-hand side of the conversion could be represented in the following HOAS representation.

$$\textbf{type}\ D_{\mathrm{ex}}\ f\ \Delta\ \alpha s\ \tau = Env\ f\ \Delta \to (Env\ f\ \Delta, Env\ f\ \alpha s, f\ \tau)$$

With this type, the implementation of *letrD* would be easy:

$$\mathit{letrD}_{\mathrm{ex}} :: (f\ a \to D_{\mathrm{ex}}\ f\ \Delta\ (\alpha : \alpha s)\ \tau) \to D_{\mathrm{ex}}\ f\ (\alpha : \Delta)\ \alpha s\ \tau$$
$$\mathit{letrD}_{\mathrm{ex}}\ h\ (\mathit{ECons}\ f_a\ \mu) = \mathbf{let}\ (\mu', \mathit{ECons}\ f'_a\ w, e') = h\ f_a\ \mu\ \mathbf{in}\ (\mathit{ECons}\ f'_a\ \mu', w, e')$$

However, the same construction is not possible if $\exists\Delta.\ D_{\mathrm{ex}}\ f\ \Delta\ \alpha s\ \tau$ is used. To bind the existential Δ, we need to apply h to some argument, but to prepare the argument, we need to choose the existential in $\mathit{letrD}_{\mathrm{ex}}$'s return value to be $\alpha : \Delta$—there is a cyclic dependency regarding the existential type Δ. Of course, there would be no issue if the existential quantification were placed outside the function as $\exists\Delta.f\ a \to D_{\mathrm{ex}}\ f\ \Delta\ (\alpha{:}\alpha s)\ \tau$. However, by definition, HOAS represents binders as functions, which must be of function types rather than existential types. Thus, in HOAS, opening existential types (and hence revealing Δ) can only be possible after applications of such functions. Note that $(P \Rightarrow \exists x.Qx) \Rightarrow (\exists x.P \Rightarrow Qx)$ is not an intuitionistic theorem (but a classical theorem), intuitively because the choice of x can depend on P for the antecedent formula. Actually, such dependency is not possible in the tagless final representation, provided that f is abstract [1], but this information is not immediately available to language implementors. The normalization based on PushSliding, which prohibits **push** from occurring after **letr**, does not help this issue of choice timing. Even after the normalization, **letr** still works on bindings and worklists, which are obtained only after opening existential types.

Nevertheless, we have workarounds. A straightforward approach is to use *Dynamic* to perform dynamic type checking, which, however, requires us to scatter *Typeable* constraints in every construct. This is not ideal as it changes parts of the EDSL syntax that are otherwise irrelevant to mutually recursive definitions. Another, slightly heavyweight approach is to use unembedding [1,2,27] to convert the tagless final representation to the de Bruijn-indexed one, so that we can implement the above conversion ($\rightsquigarrow$). The original approaches do not literally support multiple syntactic categories, but the extension would be straightforward; the implementation[7] of embedding-by-unembedding [27] experimentally supports them as of version 0.4. The conversion essentially ensures that the above-mentioned dependency is not possible.

This gap may be intrinsic: MRD builders enable a local, step-by-step construction of mutually recursive definitions without referring to the global information (i.e., mutually bound variables), but this convenience comes at the nontrivial cost of recovering this hidden information.

4 Theoretical Background: Trace Operator

In this section, we discuss the theoretical background of MRD builders: the trace operator [21].

4.1 Trace Operator

Intuitively, the trace operator can model knot-tying in functional programming. Formally, given a morphism $f \in C(A \otimes X, B \otimes X)$ in a symmetric monoidal

[7] https://github.com/kztk-m/EbU

category C, the trace operator $\mathrm{Tr}^X_{A,B}$ "traces out" X to produce a morphism $\mathrm{Tr}^X_{A,B} f \in C(A, B)$. (We only consider the symmetric case.) The trace $\mathrm{Tr}^X_{A,B}$ feeds back X, which is characterized by certain laws. Among these laws, the VANISHING law is interesting as it enables us to build a "bigger" trace operator from smaller trace operators (in terms of objects to be traced out)

$$\mathrm{Tr}^I_{A,B} f = f \qquad \mathrm{Tr}^{X\otimes Y}_{A,B} f = \mathrm{Tr}^X_{A,B}(\mathrm{Tr}^Y_{A\otimes X, B\otimes X} f) \qquad \text{(VANISHING)}$$

suggesting that the trace operator lets us define constructs for mutually recursive definitions that compose single definitions one by one. The second equation, in particular, is where the definitions of *letr* in Section 2.3 come from.

4.2 Design Principles of MRD Builders Based on Trace Operator

A common approach to view $\Gamma \vdash e : \tau$ as a morphism in a certain monoidal category C is to model it as a morphism from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ in C, where $[\![\Gamma]\!]$ is defined as $[\![x_1 : \sigma_1, \ldots, x_n : \sigma_n]\!] = [\![\sigma_1]\!] \otimes \cdots \otimes [\![\sigma_n]\!]$. In this view, a trace operator can naturally be modeled as:

$$\frac{\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \tau \times \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash \mathbf{tr}\, x_1 \ldots x_n . e : \tau}$$

Here, $\times$ is a type operator to be interpreted by the tensor product $\otimes$ in C.

This straightforward approach, however, has some issues when we use it in EDSL design. First, since **tr** can be decomposed into smaller traces by the VANISHING law, the core syntax need only support the simple form with $n = 1$ above, reducing the implementation effort. Second, some EDSLs lack a product type suitable for **tr**, as illustrated by the grammar example below. Consider context-free grammars with semantic actions. If we ignore nonterminals, a grammar is simply represented in the tagless final style as $(\mathit{Applicative}\ f, \mathit{Alternative}\ f, \mathit{LiftChar}\ f) \Rightarrow f\ a$, where *LiftChar* is a type class with a method $\mathit{fromChar} :: \mathit{Char} \to f\ \mathit{Char}$ that represents a terminal symbol. *Applicative* and *Alternative* are standard-library type classes, which represent concatenations and unions in this setting, respectively. This EDSL has a product (a, b), but $f\ (a, b)$ represents a grammar whose action returns a pair, not a pair of grammars. As a result, a single recursive definition of $f\ (a, b)$ gives us a nonterminal, instead of two nonterminals. In other words, in this case, we *want* to interpret Γ as a bundle of nonterminals, but the EDSL has no type for this.

Accordingly, we prepare a variant **letr** of **tr** that focuses only on a single variable, and a special syntactic category d so as not to interfere with the existing EDSL expressions, as shown in Fig. 1. The intuition underlying d is that the builder-in-context $\Gamma \vdash d : \sigma_1, \ldots, \sigma_n \rhd \tau$ permits an interpretation in $C([\![\Gamma]\!], [\![\tau]\!] \otimes [\![\sigma_n]\!] \otimes \cdots \otimes [\![\sigma_1]\!])$ so that $\sigma_1, \ldots, \sigma_n$ are to be traced out. We guard the τ element from being traced out so that we finally get $C([\![\Gamma]\!], [\![\tau]\!])$, guaranteeing that we can return to the original interpretation domain of EDSL expressions. Adding a new syntactic category is straightforward in the tagless final style [9], as demonstrated by the *Defs* type class in Section 2.1.

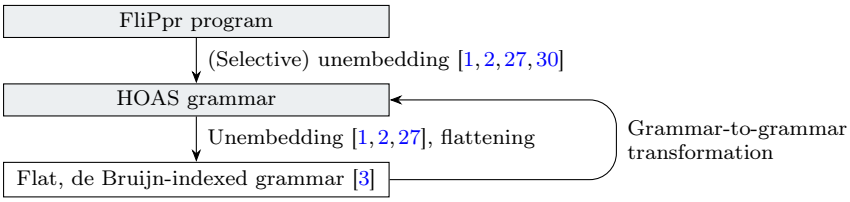

Fig. 2: Architecture of FliPpr: the highlighted parts use MRD builders

As a result, our proposed constructs can be interpreted straightforwardly in a traced monoidal category C, assuming diagonal maps $\delta \in C(A, A \otimes A)$.

$$\begin{aligned}
[\![\Gamma \vdash \mathbf{ret}\ e : \triangleright\tau]\!] &= [\![\Gamma \vdash e : \tau]\!] \\
[\![\Gamma \vdash \mathbf{push}\ e\ d : \sigma, \overline{\sigma} \triangleright \tau]\!] &= ([\![\Gamma \vdash d : \overline{\sigma} \triangleright \tau]\!] \otimes [\![\Gamma \vdash e : \sigma]\!]) \circ \delta \\
[\![\Gamma \vdash \mathbf{letr}\, x.d : \overline{\sigma} \triangleright \tau]\!] &= \mathrm{Tr}^{[\![\sigma]\!]} [\![\Gamma, x : \sigma \vdash d : \sigma, \overline{\sigma} \triangleright \tau]\!] \\
[\![\Gamma \vdash \mathbf{freeze}\ d : \tau]\!] &= [\![\Gamma \vdash d : \triangleright\tau]\!]
\end{aligned}$$

A caveat is that, in general, we require additional effort [1,2,27] to implement the semantics due to the use of Γ, hidden in HOAS; we suspect this cost is intrinsic to giving nontrivial semantics to HOAS. The same effort suffices for converting MRD builders to **letrec** (Section 3.2), though for a different reason.

A traced monoidal category comes with other laws besides the vanishing law. Among them, the superposing law below is worth mentioning, as it is related to the PushSliding law discussed in Section 2.1. In other words, the PushSliding law is not arbitrary but has an origin in the theoretical background.

$$g \otimes \mathrm{Tr}^X_{A,B} f = \mathrm{Tr}^X_{C \otimes A, D \otimes B} (g \otimes f)$$

This, combined with other laws, yields the PushSliding law.

5 Applications in FliPpr

In this section, we report the uses of MRD builders in the embedded version [30] of FliPpr [29], an invertible [28] pretty-printing system. In FliPpr, users write pretty-printers using tailored pretty-printing combinators [40] under a restriction, so that it can generate context-free grammars for the printer, with the guarantee that pretty-printed strings are always correctly parsed.[8] FliPpr uses the MRD builders in two places: as a surface language that users write, and as target grammars of grammar transformations (Fig. 2).

5.1 Surface Use in FliPpr

The surface use follows *mkEvenOdd2* in Section 2.3, which leverages `RecursiveDo`, with two notable improvements. First, since FliPpr adopts a type-constructor-headed expression type, specifically, PHOAS [11], it does not require W/unW-like

[8] When multiple parsing results exist, the system guarantees that one of the results corresponds to the AST that is pretty-printed.

wrappers, by having a custom instance of *RecArg* for its expression type. We note that we can still reuse the W-guarded instance via the `DerivingVia` extension [5], as below, where we write *Exp* v for the FliPpr expression type constructor (v is an abstract type coming from the PHOAS encoding).

deriving via W (*Exp* v τ) **instance** *RecArg* (*DefM* (*Exp* v)) (*Exp* v τ)

Second, we also have a *RecArg* instance for functions of type *In* v $a_1 \to \cdots \to$ *In* v $a_n \to$ *Exp* v τ,[9] allowing recursive bindings of functions without *lam*/*app*.

This adoption of MRD builders departs from the original embedded FliPpr [30], which implements mutually recursive definitions via references. The reference-based approach was error-prone in internal processing, specifically causing non-termination and the circular evaluation error, which motivated MRD builders.

5.2 Internal Use in FliPpr

Another important use of *letr1* appears in FliPpr's internal grammar-to-grammar transformations (Fig. 2), where MRD builders are used for target grammars. We do not use them for source grammars, because HOAS does not support intensional analysis well. Thus, before transformation, source grammars are converted to a flat (no nested recursive definitions), first-order (namely, de Bruijn-indexed) representation [3].

Internally, FliPpr transforms a grammar by applying a finite-state transducer [31]. We refer the reader to [31] for its motivation; here we focus on the procedure, which is essentially a form of on-demand product construction: it generates nonterminals N_{q_1,q_2} from original nonterminals N and transducer states q_1 and q_2, where N_{q_1,q_2} produces the strings obtained by running the transducer from state q_1 to state q_2 over the strings produced by N. The procedure traverses grammar expressions, parameterized by states q_1 and q_2, expanding nonterminals adaptively. Specifically, on encountering a nonterminal N, it proceeds as follows: (1) if N_{q_1,q_2} is already generated, return it; (2) otherwise, introduce a fresh N_{q_1,q_2} and record it as generated; (3) process its right-hand side R to obtain R_{q_1,q_2}; and (4) add a new rule $N_{q_1,q_2} ::= R_{q_1,q_2}$, and return N_{q_1,q_2}.

Fig. 3 shows the relevant code in FliPpr,[10] where *letr1* performs this dynamic generation; x and a correspond to N and N_{q_1,q_2} above, respectively. *DefM* is used with *StateT* to track processed nonterminals. The code looks complicated because we need to peel off *StateT* to apply *letr1*, which works on *DefM*, but it faithfully implements the procedure above.[11] We note that *letr1* is also used for simple inlining of grammars; we omit the details as its use is similar.

[9] *In* v is a function argument type in FliPpr; since FliPpr is a first-order language, function arguments and results are distinguished by types.

[10] https://github.com/kztk-m/flippre/blob/ec316746d6b5e8b646448915fcd91d1dc8ec59a6/flippre-backend-grammar/src/Text/FliPpr/Grammar/ExChar.hs

[11] More idiomatic programming is possible if we generalize *letr1* to work with a class of monads other than *DefM* f.

```
-- IxN env a: a nonterminal in a flat grammar (de Bruijn index)
-- env: types of nonterminals in a flat grammar
-- Qsp: a transducer state
-- lookupMemo :: Memo env g → Qsp → Qsp → IxN env a → Maybe (g a)
-- updateMemo :: Memo env g → Qsp → Qsp → IxN env a → g a → Memo env g
procVar :: ··· ⇒ Qsp → Qsp → IxN env a → StateT (Memo env g) (DefM g) (g a)
procVar q1 q2 x = StateT $ λmemo →
  case lookupMemo memo q1 q2 x of
    Just r → return (r, memo)
    Nothing → do
      let rhs = lookIxMap defsMap x
      letr1 $ λa → do
        (r, memo′) ← runStateT (procRHS q1 q2 rhs) (updateMemo memo q1 q2 x a)
        return (r, (a, memo′))
```

Fig. 3: Use of *letr1* in internal grammar processing in FliPpr

Performance Note. A naive implementation has two severe bottlenecks. One is the tally representation of de Bruijn indices, which is easily addressed by representing them as *Word*s via a phantom type. Another bottleneck is the composition of shifting functions during flattening (Fig. 2), which works on de Bruijn-indexed terms after unembedding [1,2,27] and generates one big **letrec**. Since this flattening tends to compose simple unconditional shifting functions to the left, we use a designated datatype that carries the total amount of such shifts. With these workarounds, processing time for a fairly large grammar (with >10,000 nonterminals after flattening) is reduced from 40 minutes to 3 seconds.[12] Although the flattening is an extreme case in terms of shifting, we expect a similar technique to be useful for the conversion in Section 3.2.

6 Discussion and Related Work

The *letrec* combinator in Section 1 mirrors the standard simple typing rule for **letrec**. (A tupled [10,19] variant is also possible.) This *letrec* takes the shape (Δ) information as the first argument, which is crucial for some interpretations such as those that use initial values to compute a fixed point. Otherwise, since the shape of Δ can only be known by pattern matching values of type $Env\ f\ \Delta$ for some f (which are unavailable during the interpretation), the possible interpretations would be limited to tying the knot. For such cases, there is no strong reason to have EDSL-level mutually recursive definitions. A similar representation is used by Oliveira and Löh [37], with a type class constraint replacing $Env\ Proxy\ \Delta$. Kiselyov [22] avoids the shape parameter by applying the converse of the tupling for the ($Env\ Exp\ \Delta \to Env\ Exp\ \Delta$) part, which then carries the shape information. This representation, however, makes the size of the recursive bindings

[12] The experiments were conducted on MacBook Pro (14-inch, 2021) with Apple M1 Max CPU and 32 GB of memory, compiled using GHC 9.6.7 with `-O1`.

quadratic in the number of such bindings. We can also use a first-order de Bruijn-indexed representation for **letrec**. Baars et al. [3] adopted such a first-order representation for typed context-free grammars. However, writing and generating programs are laborious with de Bruijn indices, although some interpretations are known to be difficult when we use the HOAS representation instead [1,2,27].

We need not use heterogeneous lists (*Env*) to represent **letrec**. Devriese and Piessens [13] and Brink et al. [6] use the following fixed-point combinator.

$$kfix :: ((\forall a.\, Key\ a \to Exp\ a) \to (\forall a.\, Key\ a \to Exp\ a)) \to Key\ a \to Exp\ a$$

By choosing an appropriate GADT for the *Key* type, where each constructor $C\ \tau$ indicates a binding of type τ, we can define mutually recursive bindings.

Rips et al. [36] use the combinator $def :: Def\ a\ f \Rightarrow (a \to (a, f\ b)) \to f\ b$ for recursive definitions. Although its type resembles our $letr :: RecArg\ m\ t \Rightarrow (t \to m\ (t, r)) \to m\ r$, there is a critical difference: our *letr* is a *derived* interface, while their *def* is a core primitive. Being a core primitive, *def* has a type that is too general without restricting a; as a ranges over the host's types, the only possible interpretation would be tying the knot, i.e., $def\ f = \mathbf{let}\ (a, r) = f\ a\ \mathbf{in}\ r$. Thus, appropriate restrictions of a are needed to have non-trivial interpretations, whereas they address this in an ad hoc manner for each interpretation (e.g., printing). Another crucial difference: *def* takes a pure function and returns a single EDSL expression ($f\ b$), while *letr* takes a monadic function and returns an arbitrary Haskell value (of type r) in a monad. As we demonstrated in Section 2.2, this difference is crucial for both flexible programming and dynamic generation.

Yallop and Kiselyov [41] discuss an interface for generating mutually recursive definitions in metaprogramming; this approach is then simplified [24] and implemented in MetaOCaml. In Haskell notation, their combinators [24] can be written as $withLocusRec :: (Locus \to Code\ \alpha) \to Code\ \alpha$ and $mkGenLet :: Eq\ k \Rightarrow Locus \to (k \to Code\ (\alpha \to \beta)) \to (k \to Code\ (\alpha \to \beta))$. Intuitively, the first call of $mkGenLet\ loc\ f\ k$ yields a fragment of a mutually recursive definition $f_k = f\ k$ to be inserted into **letrec** at the *withLocusRec* invocation that generated *loc*, and the second call of $mkGenLet\ loc\ f\ k$ will be replaced with f_k. The biggest difference is that they generate **letrec**, while we do not; we instead generate MRD builder constructs. Here is a quote from their paper [41]: "*The limited support for generating mutual recursion is a consequence of expression-based quotation: brackets enclose expressions, and splices insert expressions into expressions—but a group of bindings is not an expression.*" From this perspective, we sidestepped the difficulty by considering an additional syntactic category d in Fig. 1 besides expressions. They also provide an implementation of these combinators for EDSLs in OCaml without destructive updates.[13] This, however, involves many unsafe type coercions (`Obj.magic`); this stems from their denotational semantics [24] on which it is based; the semantics throws away type information in various places, such as free variables in generated code. A well-typed implementation for their interface that supports heterogeneous mutually recursive definitions still remains open. In contrast, as we have shown in Section 2, we can implement

[13] https://okmij.org/ftp/meta-programming/genletrec/index.html#impl

our combinators *local* and *letr1* without *unsafeCoerce*. However, it is fair to note that the translation of our constructs to **letrec** in Section 3.2 requires a conversion [1, 2, 27] to de Bruijn-indexed terms, whose known implementations involve runtime type-checking or *unsafeCoerce* in coercing variables $\Gamma \vdash x : \sigma$ to $\Gamma' \vdash x : \sigma$. Nevertheless, such seemingly-unsafe coercions are proven to be safe by Kripke parametricity, which proves that Γ' must always have the form Γ, Γ'' for some Γ''; in other words, the coercions must be weakening operations.

The EDSL `accelerate` reifies implicit sharing into EDSL-level **let** [33], exploiting the fact that GHC evaluates cyclic first-order data into cyclic structures in memory. For example, **let** $ones = 1 : ones$ **in** $ones$ yields a cons cell whose cdr points to itself. Such pointers can be compared via *StableName* in GHC, enabling us to detect graph structures in memory [18]. Although the original method [33] targets non-recursive **let** instead of **letrec**, we conjecture that the method can be extended to recursive **let**. This approach is extremely simple for surface programming: particularly, it allows top-level recursive definitions. However, it is error-prone if we programmatically generate EDSL expressions.

Unembedding [1] provides an interconversion between (parametric/tagless-final) HOAS and de Bruijn-indexed representations of simply-typed ASTs, which is provably correct based on Kripke parametricity. An advantage of the conversion to de Bruijn-indexed representations is its support for intensional analysis of ASTs, for example, for optimization [2], while keeping the programming flexibility provided by the HOAS representation. Another advantage is its support for interpretation of open terms [27], which is crucial for incremental computation [7, 17] and bidirectional transformations [16, 32]. Section 3.2 highlights another use case: supporting existentially-quantified semantic domains, for which the HOAS representation changes the timing of choice for existentials.

7 Conclusion

In this paper, we proposed MRD builders, whose usefulness lies especially in EDSL program generation. The idea of the constructs is inspired by the trace operator in category theory. We presented their syntax and typing rules in a simply-typed system and showed their HOAS representations. The proposed constructs are interconvertible with **letrec**, although converting to **letrec** requires an additional conversion to de Bruijn-indexed representation. We described the surface and internal use of MRD builders in the invertible pretty-printing system FliPpr.

The artifact for this paper is available at https://doi.org/10.5281/zenodo.20431647, which includes a minimal implementation of MRD builders, and FliPpr snapshots to support the discussions in Section 5.2.

Acknowledgments. We thank the anonymous TFP 2026 reviewers for their helpful feedback. This work was partially supported by JSPS KAKENHI Grant Numbers JP20H04161, JP23K20379, JP22H03562 and JP23K24818.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Atkey, R.: Syntax for free: Representing syntax with binding using parametricity. In: Curien, P. (ed.) Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5608, pp. 35–49. Springer (2009). https://doi.org/10.1007/978-3-642-02273-9_5
2. Atkey, R., Lindley, S., Yallop, J.: Unembedding domain-specific languages. In: Weirich, S. (ed.) Haskell. pp. 37–48. ACM (2009). https://doi.org/10.1145/1596638.1596644
3. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed grammars: The left corner transform. Electron. Notes Theor. Comput. Sci. **253**(7), 51–64 (2010). https://doi.org/10.1016/j.entcs.2010.08.031
4. Bekić, H.: Definable operation in general algebras, and the theory of automata and flowcharts. In: Jones, C.B. (ed.) Programming Languages and Their Definition - Hans Bekic (1936-1982). pp. 30–55. Lecture Notes in Computer Science, Springer (1984). https://doi.org/10.1007/BFB0048939
5. Blöndal, B., Löh, A., Scott, R.: Deriving via: or, how to turn hand-written instances into an anti-pattern. In: Wu, N. (ed.) Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018. pp. 55–67. ACM (2018). https://doi.org/10.1145/3242744.3242746
6. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6120, pp. 58–79. Springer (2010). https://doi.org/10.1007/978-3-642-13321-3_6
7. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 145–155. ACM (2014). https://doi.org/10.1145/2594291.2594304
8. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Sci. Comput. Program. **76**(5), 349–375 (2011). https://doi.org/10.1016/J.SCICO.2008.09.008
9. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205
10. Chin, W.: Towards an automated tupling strategy. In: Schmidt, D.A. (ed.) Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993. pp. 119–132. ACM (1993). https://doi.org/10.1145/154630.154643
11. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Hook, J., Thiemann, P. (eds.) ICFP. pp. 143–156. ACM (2008). https://doi.org/10.1145/1411204.1411226
12. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170
13. Devriese, D., Piessens, F.: Finally tagless observable recursion for an abstract grammar model. J. Funct. Program. **22**(6), 757–796 (2012). https://doi.org/10.1017/S0956796812000226

14. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Voigtländer, J. (ed.) Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012. pp. 117–130. ACM (2012). https://doi.org/10.1145/2364506.2364522
15. Fegaras, L., Sheard, T.: Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In: Boehm, H., Jr., G.L.S. (eds.) Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. pp. 284–294. ACM Press (1996). https://doi.org/10.1145/237721.237792
16. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007)
17. Giarrusso, P.G., Régis-Gianas, Y., Schuster, P.: Incremental λ-calculus in cache-transfer style - static memoization by program transformation. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11423, pp. 553–580. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_20
18. Gill, A.: Type-safe observable sharing in haskell. In: Weirich, S. (ed.) Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009. pp. 117–128. ACM (2009). https://doi.org/10.1145/1596638.1596653
19. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: Peyton Jones, S.L., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997. pp. 164–175. ACM (1997). https://doi.org/10.1145/258948.258964
20. Huet, G.P., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Inf. **11**, 31–55 (1978). https://doi.org/10.1007/BF00264598
21. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society **119**(3), 447–468 (Apr 1996)
22. Kiselyov, O.: Simplest poly-variadic fixpoint combinators for mutual recursion. https://okmij.org/ftp/Computation/fixed-point-combinators.html#Poly-variadic (2002), last visit: 2025-12-10
23. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Nilsson, H. (ed.) Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004. pp. 96–107. ACM (2004). https://doi.org/10.1145/1017472.1017488
24. Kiselyov, O., Yallop, J.: let (rec) insertion without effects, lights or magic. CoRR **abs/2201.00495** (2022). https://doi.org/10.48550/arXiv.2201.00495
25. Kock, A.: Continuous yoneda representations of a small category. Preprint (1966), available on http://tildeweb.au.dk/au76680/CYRSC.pdf
26. Kovács, A.: Closure-free functional programming in a two-level type theory. Proc. ACM Program. Lang. **8**(ICFP), 659–692 (2024). https://doi.org/10.1145/3674648
27. Matsuda, K., Frohlich, S., Wang, M., Wu, N.: Embedding by unembedding. Proc. ACM Program. Lang. **7**(ICFP), 1–47 (2023). https://doi.org/10.1145/3607830

28. Matsuda, K., Mu, S.C., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: Gordon, A.D. (ed.) ESOP. Lecture Notes in Computer Science, vol. 6012, pp. 448–467. Springer (2010)
29. Matsuda, K., Wang, M.: FliPpr: A prettier invertible printing system. In: Felleisen, M., Gardner, P. (eds.) ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 101–120. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_6
30. Matsuda, K., Wang, M.: Embedding invertible languages with binders: a case of the FliPpr language. In: Wu, N. (ed.) Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018. pp. 158–171. ACM (2018). https://doi.org/10.1145/3242744.3242758
31. Matsuda, K., Wang, M.: Flippr: A system for deriving parsers from pretty-printers. New Gener. Comput. **36**(3), 173–202 (2018). https://doi.org/10.1007/S00354-018-0033-7
32. Matsuda, K., Wang, M.: HOBiT: Programming lenses without using lens combinators. In: Ahmed, A. (ed.) ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 31–59. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_2
33. McDonell, T.L., Chakravarty, M.M.T., Keller, G., Lippmeier, B.: Optimising purely functional GPU programs. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 49–60. ACM (2013). https://doi.org/10.1145/2500365.2500595
34. Miller, D., Nadathur, G.: A logic programming approach to manipulating formulas and programs. In: Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987. pp. 379–388. IEEE-CS (1987)
35. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988. pp. 199–208. ACM (1988). https://doi.org/10.1145/53990.54010
36. Rips, B.M., Janssen, N., Lubbers, M., Koopman, P.: Shallowly embedded functions. In: Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming. PPDP '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3756907.3756923
37. d. S. Oliveira, B.C., Löh, A.: Abstract syntax graphs for domain specific languages. In: Albert, E., Mu, S. (eds.) Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013. pp. 87–96. ACM (2013). https://doi.org/10.1145/2426890.2426909
38. Swadi, K.N., Taha, W., Kiselyov, O., Pasalic, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: Hatcliff, J., Tip, F. (eds.) Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006. pp. 160–169. ACM (2006). https://doi.org/10.1145/1111542.1111570
39. Voigtländer, J.: Asymptotic improvement of computations over free monads. In: Audebaud, P., Paulin-Mohring, C. (eds.) Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5133, pp. 388–403. Springer (2008). https://doi.org/10.1007/978-3-540-70594-9_20
40. Wadler, P.: A prettier printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, chap. 11. Palgrave Macmillan (2003)

41. Yallop, J., Kiselyov, O.: Generating mutually recursive definitions. In: Hermenegildo, M.V., Igarashi, A. (eds.) Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019. pp. 75–81. ACM (2019). https://doi.org/10.1145/3294032.3294078