Mutually Recursive Definition Builders
(A Working Draft for Discussion)

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 we want to perform intensional analysis or manipulation of recursive definitions. A common approach to model mutually recursive definitions is recursive **let** (**letrec**). However, in the typed setting, **letrec** prevents us from constructing recursive definitions locally step-by-step, which is inconvenient in 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 a set of language constructs that we call *mutually recursive definition builders*, which enable local and step-by-step 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 report our experience using the proposed constructs in the embedded 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 a general-purpose language. An embedded domain-specific language (EDSL), which is a DSL implemented as a library in a host language, provides extra convenience for both DSL programmers and implementers, as they can access the host language's ecosystem, such as parsers, a type system, editor support, an efficient compiler, and interoperability with the host language.

Recursive definitions are one of the fundamental concepts in programming. This also applies to EDSLs. Although we often do not need to handle them explicitly in many applications for which using the host's recursive definitions is sufficient, explicit handling 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* [10, 29].

For defining a single recursion, using the standard fixed-point combinator suffices. When we implement it as an EDSL primitive, if we use higher-order abstract syntax (HOAS) [9, 15, 26, 27] with an expression type *Exp*, its type is given as $\mathit{fix} :: (\mathit{Exp}\ a \to \mathit{Exp}\ a) \to \mathit{Exp}\ a$.[1][2] But, what can we do with *mutual recursion*? Theoretically, the operator *fix* is known to be powerful enough to support mutual recursions via Bekič's lemma.

$$\mathit{fix2} :: ((\mathit{Exp}\ a, \mathit{Exp}\ b) \to (\mathit{Exp}\ a, \mathit{Exp}\ b)) \to (\mathit{Exp}\ a, \mathit{Exp}\ b)$$
$$\mathit{fix2}\ f = (\mathit{fix}\ \$\ \lambda x \to \mathit{fst}\ \$\ f\ (x, \mathit{fix}\ \$\ \lambda y \to \mathit{snd}\ \$\ f\ (x, y)), \dots)$$

However, the recursive definitions realized in this way are not useful due to the code-size blow-up. In general, we need 2^n copies of recursive definitions for n mutual definitions, which is impractical.

Thus, a practical EDSL must be equipped with an operator to define 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]

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

Here, $\mathit{Env}\ f\ \Delta$ is a heterogeneous list [18] of expressions of type Δ, namely, $\mathit{Env}\ f\ [a_1, \dots, a_n] \simeq (f\ a_1, \dots, f\ a_n)$. Thus, it bundles the following combinators.

$$\mathit{letrec}_0 :: (() \to ()) \to (() \to \mathit{Exp}\ t) \to \mathit{Exp}\ t$$
$$\mathit{letrec}_1 :: (\mathit{Exp}\ a \to \mathit{Exp}\ a) \to (\mathit{Exp}\ a \to \mathit{Exp}\ t) \to \mathit{Exp}\ t$$
$$\mathit{letrec}_2 :: ((\mathit{Exp}\ a, \mathit{Exp}\ b) \to (\mathit{Exp}\ a, \mathit{Exp}\ b)) \to ((\mathit{Exp}\ a, \mathit{Exp}\ b) \to \mathit{Exp}\ t) \to \mathit{Exp}\ t$$
$$\dots$$

There also exist other variants [3, 4, 10, 17, 29].

However, to the best of our knowledge, the existing approaches [3, 4, 10, 17, 29] share the common issue: we need to be explicit about the objects to be defined mutually recursively in advance, especially Δ. This "global" construction of mutually recursive definitions 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 and the functions that call it but also the rest of the functions as Δ changes. Also, the exposure of Δ makes

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

[2] For presentation simplicity, we often present types of combinators in the plain HOAS, while the plain HOAS has an issue with interpretations [11] due to exotic terms [8] and there exist HOAS representations [6, 8] without the issue. If we were to use such representations, *fix* would be a constructor $\mathit{Fix} :: (v\ a \to \mathit{Exp}\ v\ a) \to \mathit{Exp}\ v\ a$ (in the parametric HOAS [8]) or a typeclass method $\mathit{fix} :: \mathit{Exp}\ f \Rightarrow (f\ a \to f\ a) \to f\ a$ (in the tagless-final style [6]).

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

```
-- A surface (derived) combinator we propose for a recursive definition of f a.
letr1 :: Defs f ⇒ (f a → DefM f (f a, r)) → DefM f r
-- A construction of recursive bindings corresponding to
-- letrec {x_i = x_((i mod n)+1)}_(1≤i≤n) in x_1
example :: Defs f ⇒ Int → DefM f (f a)
example n = fmap (!!0) (go n [])
  where
    go 0 xs = pure xs
    go i xs = letr1 $ λx_i → do
      -- xs_final is a list of bound variables x_1, ..., x_n.
      xs_final ← go (i − 1) (x_i : xs)
      pure (xs_final !! ((i `mod` n) + 1), xs_final)
```

Fig. 1: Example of generation of recursive bindings

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 we cannot know how many nonterminals will be generated prior to a transformation (otherwise, the generation is not on-demand), it is difficult or at least nontrivial to realize the transformation via *letrec*-like combinators. Thus, we aim to provide a more fine-grained construction of mutually recursive definitions, enabling one to work with a single binding at a time without referring to the global binding information.

In this paper, we propose an approach called *mutually recursive definition builders* (MRD builders, for short) to construct mutually recursive definitions from single bindings without knowing the whole bindings (the Δ above). A core idea is to have an additional syntactic category d that has a type of the form $\tau_1, \ldots, \tau_n \rhd \tau$ representing a partially constructed **letrec** expression of type τ together with a work list of expressions of types $\tau_1, \ldots, \tau_n$ to be named and put in the **letrec** bindings. With our proposed constructs, a monolithic expression **let rec** $\overline{x = e}$ **in** e' is broken down into the following compound form

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

enabling us to construct mutually recursive definitions one-by-one. Another strength of this approach is that, leveraging the power of tagless-final representation [6], MRD builders can be implemented as an "extension kit" for an existing non-recursive EDSL, without invading EDSL types. 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 [8] and the tagless final style [6]. This design also enables us to derive EDSL-independent combinators, which eases generation of EDSL expressions. As a demonstration of its expressive power, Fig. 1 demonstrates a construction of mutually recursive definitions with a derived combinator *letr1*, where the type of recursively bound variables (Δ)

is determined dynamically; this example appears to be contrived, but serves as a miniature of the practical use we will demonstrate later (Section 5.3). Although the code in Fig. 1 relies on the partial operation (!!), *letr1* guarantees well-scopedness and well-typedness of the resulting program for successful runs; for this particular program, actually (!!) does not fail if $n > 0$.

In summary, our contributions are.

- We propose combinators for mutually recursive definitions that enable the "local" construction of each object to be defined mutually (Section 2.1). We also provide several derived combinators to make programming with the proposed combinators easier (Section 2.2).
- 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, although the conversion from **letrec** (i.e., **letrec** on top of MRD builders) remains easy (Section 3.1), we need additional effort [1, 2, 20] to support the opposite direction (MRD builders on top of **letrec**) due to the implicit nature of the whole binding type (Section 3.2).
- We discuss the theoretical background of the proposed combinators, which also demonstrates the feasibility of the proposed approach towards certain semantics (Section 4). More specifically, our design of the proposed constructs is inspired by the trace operator in category theory [16]. This further highlights that traced monoidal categories can be used as the semantic domain for EDSLs that use our combinators.
- We report our implementation of the proposed idea in the embedded version [22] of FliPpr [21, 23], a language for invertible pretty-printers, and discuss non-trivial use cases (Section 5).

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

As explained in Section 1, our idea is to have an additional syntactic category representing MRD builders, which intuitively denotes a partially constructed **letrec**. Although our focus is on embedded DSLs, we first illustrate our proposed constructs in the context of a standalone DSL, as presented in Fig. 2. As we have seen in Section 1, with them, we can break $\mathbf{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathbf{in}\ e'$ down to compound form $\mathbf{freeze}\ \big(\mathbf{letr}\ x_1. \ldots \mathbf{letr}\ x_n. \mathbf{push}\ e_n\ (\ldots(\mathbf{push}\ e_1\ (\mathbf{ret}\ e'))\ldots)\big)$. 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 work list (more precisely, a stack) of expressions $e_1, \ldots, e_n$ where each e_i has type τ_i to be named and put in the **letrec** bindings. The operator $\mathbf{ret}\ e$ produces a **letrec** with the empty body, i.e., $\mathbf{letrec\ in}\ e$, together with the empty work list, $\mathbf{push}\ e\ d$ pushes e into the work list part of d, $\mathbf{letr}\ x.d$ names

Syntax

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

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

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

Fig. 2: Proposed constructs and their typing rules (as a standalone DSL)

the head e of the work list in d and moves $x = e$ to the binding part, and **freeze** d finally converts it to the standard **letrec**, provided that the work list is empty. The crucial point is that the type of already constructed bindings is hidden in the definitions, while the work list expressions can refer to hidden bindings. Of course, 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 can programmatically construct EDSL expressions.

One might notice that the above intuition (the work list and partially-constructed **letrec**) of the MRD builders suggests the following identity, provided that x does not occur free in e.

$$\textbf{push}\ e\ (\textbf{letr}\ x.\,\textbf{push}\ e_x\ d) \equiv \textbf{letr}\ x.\,\textbf{push}\ e_x\ (\textbf{push}\ e\ d)$$

Thus, there will be more than one way to construct mutually recursive definitions in general, when some definition bodies do not refer to some bound variables. 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, regard this as providing flexibility in programming when we represent our idea as an EDSL, demonstrated by the derived combinators discussed in Section 2.2.

If we use the tagless final [6] representation of HOAS [9, 15, 26, 27], these constructs can be expressed as the following methods in a type class *Defs*.

```
class Defs (f :: k → Type) where
  -- D f [τ1, ..., τn] τ represents d : 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.

2.2 Surface and Derived 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 the end of this subsection. The signature *letr1* resembles a HOAS representation of the single recursive definition **letrec** $x = e_1$ **in** e_2, which has type $(f\ \alpha \to (f\ \alpha, f\ \beta)) \to f\ \beta$. Thus, the standard use of *letr1* is similar.

```
-- let ones = one : ones in ...
letr1 $ λones → do
  -- assuming one :: Exp Int and cons :: Exp a → Exp [a] → Exp [a]
  pure (cons one ones, ... {- computation using ones -} ...)
```

However, the crucial difference between *letr1* and the standard single recursive definition lies in the type parameter r that ranges over *ordinary Haskell types* instead of *EDSL types*. Thanks to this flexibility, one can use *letr1* to define mutual recursion operators such as *letr2* and *letr3* below.

```
letr2 :: Defs f ⇒ ((f α, f β) → DefM f ((f α, f β), r)) → DefM f r
letr2 h = letr1 $ λa → letr1 $ λb → do
  -- the outer function (λa → ...) has type f α → DefM f (f α, r)
  -- the inner function (λb → ...) has type f β → DefM f (f β, (f α, r))
  ((a', b'), r) ← h (a, b)
  pure (b', (a', r))   -- Notice the inversion of the order
letr3 :: Defs f ⇒ ((f α, f β, f γ) → DefM f ((f α, f β, f γ), r)) → DefM f r
letr3 h = letr1 $ λa → letr1 $ λb → letr1 $ λc → do
  ((a', b', c'), r) ← h (a, b, c)
  pure (c', (b', (a', r)))
```

This flexibility is also beneficial to define multiple variables recursively, where the number of variables is given at runtime. For example, *letr1* can be used to produce n-many copies of bindings, by using the following derived combinator *letr1s*.

```
letr1s :: (Defs f, Eq k) ⇒
          [k] → ((k → f α) → DefM f (k → f α, r)) → DefM f r
letr1s [] h = snd <$> h (const $ error "out of bounds")
letr1s (k : ks) h = letr1 $ λf_k → letr1s ks $ λf → do
  -- f is supposed to be valid for ks
  -- f' is valid for k : ks
  (f', r) ← h $ λx → if x == k then f_k else f x
  return (f', (f' k, r))
```

One may think of *letr1* as a form of name generation, coming together with the obligation to provide its definition, with a guarantee of well-typedness and well-scopedness.

It is worth noting that, since *DefM f* is built on top of the *Defs f* interface, despite such flexibility, it does not violate the abstraction provided by an EDSL. That is, *EDSL types are separated from Haskell types*.

We can generalize the construction presented by *letr2* and *letr3* to any tuples of EDSL expressions. To make this possible, we first prepare the following type class to bundle the definitions.

```
class Monad m ⇒ LetRecArg m t where
  -- both t and r range over Haskell types (i.e., t, r :: Type)
  letr :: (t → m (t, r)) → m r
```

Then, we define several instances.

```
-- Identity is defined as newtype Identity a = Identity a
instance Defs f ⇒ LetRecArg (DefM f) (Identity (f a)) where ...
instance (LetRecArg m a, LetRecArg m b)
         ⇒ LetRecArg m (a, b) where ...
instance (LetRecArg m a, LetRecArg m b, LetRecArg m c)
         ⇒ LetRecArg m (a, b, c) where ...
...
-- In the following instances, letr is defined similarly to
-- letr1s [minBound .. maxBound].
instance (LetRecArg m a) ⇒ LetRecArg m (Bool → a) where
instance (LetRecArg m a) ⇒ LetRecArg m (Word8 → a) where ...
instance (LetRecArg m a) ⇒ LetRecArg m (Int8 → a) where ...
```

A subtlety is that, if we were to declare an instance of *LetRecArg* (*DefM f*) (*f a*), it would overlap with other instance declarations to confuse Haskell instance resolution, reducing the usability. Thus, for this type class to be useful, we require users to define the base case for their specific expression type, say *Exp*, by using `DerivingVia`.

```
deriving via Identity (Exp a) instance LetRecArg (DefM Exp) (Exp a)
```

An advantage of the type class *LetRecArg* is that we can have a variant of *mfix* so that we can use Haskell-level recursive definitions to achieve EDSL-level

recursive definitions. Specifically, with *LetRecArg*, we can define the following *mfix*-like operator.

```
mfix' :: LetRecArg m t ⇒ (t → m t) → m t
mfix' h = letr $ λt → h t >>= λt' → pure (t', t)
```

We cannot make *DefM f* a *MonadFix* instance due to the type mismatch (t is unconstrained in *mfix*), which is legitimate as not all Haskell-level recursive definitions correspond to EDSL recursive definitions. However, we still can use *mfix'* to rebind `RecursiveDo` by using `QualifiedDo` or `RebindableSyntax`.

```
-- F exports mfix = mfix' together with the standard monad operations.
F.do rec f ← ...
        g ← ...
     ...
```

Thus, we can use Haskell's recursive definition syntax for EDSL's recursive definitions.

The *local* enables us to use a term that depends on recursive definitions in other terms. For example, we can write:

```
-- We assume some Defs instance Exp with appropriate
-- zero :: Exp Int, one :: Exp Int, and cons :: Exp a → Exp [a] → Exp [a]
onesM :: DefM Exp (Exp Int)
onesM = F.do rec ones ← cons one ones
                 pure ones
zeroThenOnes :: Exp [Int]
zeroThenOnes = cons zero (local onesM)
```

However, the combinator must be used with great care as it copies recursive bindings inside.

```
-- We assume zip :: Exp [a] → Exp [b] → Exp [(a, b)]
oneones :: Exp [(Int, Int)]
oneones =
  -- The following expression corresponds to:
  --   zip (letrec ones = one : ones in ones)
  --       (letrec ones = one : ones in ones)
  zip (local onesM) (local onesM)
```

It is generally better to write:

```
-- The following corresponds to:
--   letrec ones = one : ones in zip ones ones.
oneones' :: Exp [(Int, Int)]
oneones' = local $ do ones ← onesM
                      pure $ zip ones ones
```

Thus, in practice, the users would want to use it only in the last step.

We conclude the section by showing the definition of the monad *DefM* and its operations *letr1* and *local*. First, the definition of *DefM* is analogous to the codensity monad [19, 30].[4]

```
newtype DefM f a = DefM (∀αs τ. (a → D f αs τ) → D f αs τ)
unDefM :: DefM f a → (a → D f αs τ) → D f αs τ
unDefM (DefM h) = h
```

We omit its *Functor*, *Applicative*, and *Monad* instance declarations, as they are straightforward. We are now ready to show the definitions of *letr1* and *local*.

```
letr1 :: Defs f ⇒ (f α → DefM f (f α, r)) → DefM f r
letr1 h = DefM $ λk → letrD $ λa → unDefM (h a) $ λ(b, r) → pushD b (k r)
local :: Defs f ⇒ DefM f (f α) → f α
local m = freezeD $ unDefM m retD
```

3 Relationship to letrec

In this section, we discuss the relationship between our proposed constructs and **letrec** $\overline{x = e}$ **in** e'. That is, we can interconvert the two syntaxes. While the conversion from **letrec** can be performed on HOAS representations, the opposite direction is performed on the DSL syntax given in Fig. 2 rather than on their HOAS representations. This effectively means that we need a conversion from HOAS to de Bruijn-indexed terms [1, 2, 20] for the conversion to **letrec**.

3.1 Conversions from letrec: letrec on Top of Our Constructs

The conversion from **letrec** is straightforward. We just follow the intuition presented in Section 2.1. Specifically, we define the conversion by giving **letrec** a derived construct on top of our ones. With our surface interface Section 2.2, we can implement it easily.

```
data Env (f :: k → Type) (as :: [k]) where
  ENil :: Env f '[]
  (:.) :: f α → Env f αs → Env f (α : αs)
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
```

[4] We said "analogous" here as we did not confirm 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. 2. Thus, we conjecture potential functoriality.

```
letrecM (_ :. sh) h = letr1 $ λx → letrecM sh $ λxs → do
  (vvs, r) ← h (x :. xs)
  case vvs of    -- trick to tell GHC that this matching is exhaustive.
    v :. vs → pure (vs, (v, r))
```

3.2 Conversion to letrec: Our Constructs on Top of letrec

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

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

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

Note that recursive bindings will be typed as $\Gamma, \Delta \vdash \mu : \Delta$.

Recall that, in Section 2.1, we explained that d intuitively represents a partially constructed **letrec** with a work list. To perform the type-preserving translation, let us consider typing such intermediate objects. A caveat is that the work-list expressions can use the **letrec**-bound variables, and thus we instead them instead as triples $\langle \mu \,|\, \overline{e} \,|\, e \rangle$ of bindings μ, a work list $\overline{e}$, and a continuation expression e, with the following typing rule, where Δ is treated existentially in the conversion.

$$\frac{\Gamma, \Delta \vdash \mu : \Delta \quad \{\Gamma, \Delta \vdash e_i : \sigma_i\} \quad \Gamma, \Delta \vdash e : \tau}{\Delta; \Gamma \vdash \langle \mu \,|\, e_1, \ldots, e_n \,|\, e \rangle : \sigma_1, \ldots, \sigma_n \rhd \tau}$$

In particular, when the work list part is empty, we have

$$\frac{\Gamma, \Delta \vdash \mu : \Delta \quad \Gamma, \Delta \vdash e : \tau}{\Delta; \Gamma \vdash \langle \mu \,|\, |\, e \rangle : \rhd \tau}$$

which is ready to convert to $\Gamma \vdash \mathbf{letrec}\ \mu\ \mathbf{in}\ e : \tau$. This is how **freeze** is processed.

Now, we are ready to define our conversion $\Gamma \vdash d : \overline{\sigma} \rhd \tau \rightsquigarrow \Delta; \Gamma \vdash \langle \mu \,|\, \overline{e} \,|\, e \rangle : \overline{\sigma} \rhd \tau$, which is read as $\Gamma \vdash d : \overline{\sigma} \rhd \tau$ is converted to $\Delta; \Gamma \vdash \langle \mu \,|\, \overline{e} \,|\, e \rangle : \overline{\sigma} \rhd \tau$ for some Δ. We present our conversion in a type-directed style to highlight the treatment of typing environments and its type-preservation nature, whereas the conversion itself is syntax-directed.

$$\frac{}{\Gamma \vdash \mathbf{ret}\ e : \rhd \tau \ \rightsquigarrow\ \emptyset; \Gamma \vdash \langle \,|\, |\, e \rangle : \rhd \tau}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash d : \overline{\sigma} \rhd \tau \ \rightsquigarrow\ \Delta; \Gamma \vdash \langle \mu \,|\, \overline{e} \,|\, e' \rangle : \overline{\sigma} \rhd \tau}{\Gamma \vdash \mathbf{push}\ e\ d : \sigma, \overline{\sigma} \rhd \tau \ \rightsquigarrow\ \Delta; \Gamma \vdash \langle \mu \,|\, e, \overline{e} \,|\, e' \rangle : \sigma, \overline{\sigma} \rhd \tau}$$

$$\frac{\Gamma, x : \sigma \vdash d : \sigma, \overline{\sigma} \rhd \tau \ \rightsquigarrow\ \Delta; \Gamma, x : \sigma \vdash \langle \mu \,|\, e, \overline{e} \,|\, e' \rangle : \sigma, \overline{\sigma} \rhd \tau}{\Gamma \vdash \mathbf{letr}\ x.d : \overline{\sigma} \rhd \tau \ \rightsquigarrow\ \Delta, x : \sigma; \Gamma \vdash \langle \mu, x = e \,|\, \overline{e} \,|\, e' \rangle : \overline{\sigma} \rhd \tau}$$

The existential nature of Δ makes the conversion with the HOAS representation difficult; if we represent binders as functions, the timing of the choice changes. To explain this, suppose that the existential Δ were explicit. In this case, the right-hand side of the conversion could be represented in the following HOAS representation.

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

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

$$\begin{aligned} &letrD_{\text{explicit}} :: (f\ a \to D_{\text{explicit}}\ f\ \Delta\ (\alpha : \alpha s)\ \tau) \to D_{\text{explicit}}\ f\ (\alpha : \Delta)\ \alpha s\ \tau \\ &letrD_{\text{explicit}}\ h\ (f_a :\!\cdot\ \mu) = \textbf{let } (\mu', f'_a :\!\cdot\ wl, e') = h\ f_a\ \mu\ \textbf{in } (f'_a :\!\cdot\ \mu', wl, e') \end{aligned}$$

However, the same construction is not possible if $\exists\Delta.\, D_{\text{explicit}}\ f\ \Delta\ \alpha s\ \tau$ 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 the $letrD_{\text{explicit}}$'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 would appear outside the function as $\exists\Delta.f\ a \to D_{\text{explicit}}\ f\ \Delta\ (\alpha : \alpha s)\ \tau$. 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, but the information is not immediately available to language implementors.

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 the EDSL that are otherwise irrelevant to mutually recursive definitions. Another, slightly heavy-weight approach is to use unembedding [1,2,20] to convert tagless final representations to de Bruijn indexed terms, so that we can implement the above conversion ($\rightsquigarrow$). Although the original approaches do not handle multiple syntactic categories literally, we believe that it would be straightforwardly extended to the case. Actually, the implementation[5] of embedding-by-unembedding [20] experimentally supports such multiple syntactic categories as of version 0.4. We can see that the conversion essentially ensures that the above-mentioned dependency is not possible.

This gap might be intrinsic to our proposed constructs: they 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 the hidden information during conversion.

4 Theoretical Background: Trace Operator

In this section, we discuss a theoretical background of the proposed constructs, namely, the trace operator [16]. This suggests that, for a semantic domain with the

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

trace operator, this syntactic representation can be used without any non-trivial conversions [1, 2, 20] (see Section 3.1).

4.1 Trace Operator

Very roughly speaking, the trace operator can model the knot-tying in functional programming. Formally, given a morphism $f \in C(A \otimes X, B \otimes X)$ in a monoidal category C, the trace operator $\mathrm{Tr}^X_{A,B}$ "traces out" X to produce a morphism $\mathrm{Tr}^X_{A,B} f \in C(A, B)$. Intuitively, 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)

$$\begin{aligned} \mathrm{Tr}^I_{A,B} f &= f \\ \mathrm{Tr}^{X \otimes Y}_{A,B} f &= \mathrm{Tr}^X_{A,B}(\mathrm{Tr}^Y_{A \otimes X, B \otimes X} f) \end{aligned}$$

suggesting that, if we base ourselves on the trace operators, we are able to define combinators for mutual recursive definitions that can compose single definitions one-by-one. From the second equation, one might recall *letr2* defined in terms of *letr1* in Section 2.2. This equation is indeed where the idea comes from.

4.2 Design Principles

A common approach to view $\Gamma \vdash e : \tau$ as a morphism in a certain 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 a monoidal category C.

This straightforward approach, however, has some issues when we use it in EDSL design. First, since the **tr** can be decomposed into a smaller trace by the vanishing law, the core syntax must provide the simplest form, like the one with $n = 1$ in the above form, to reduce the implementation effort of its semantics. Second, for a certain EDSL to which we want to apply this technique, there is already a product type in the EDSL that interferes with this interpretation. For example, consider a parser combinator EDSL. Naturally, an expression type $P\ a$ in the EDSL represents "a parser whose result is a". This a of $P\ a$ is a Haskell type, and thus $P\ (a, b)$ represents a parser whose parsing result is a pair (a, b). However, for recursive definitions with **tr**, we require an EDSL type that represents a product of parsers. It is unrealistic to redesign the whole EDSL types to add **tr**; such surgery is not necessary for **letrec**.

Accordingly, we prepare a variant **letr** (portmanteau word of **let** and **tr**) of **tr** that focuses only on a single variable, and we introduce a special syntactic category

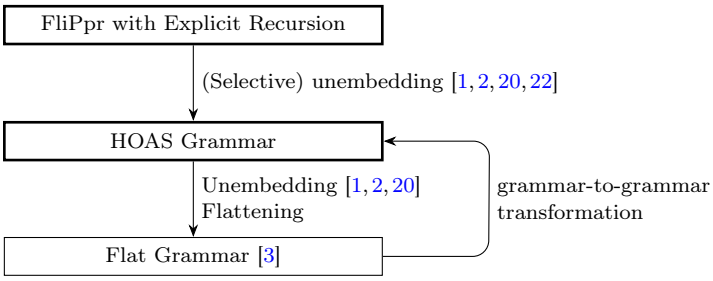

Fig. 3: Architecture of FliPpr: thick-boxed part using the proposed technique

d, as shown in Fig. 2. The intuition underlying d is that $\Gamma \vdash d : \sigma_1, \ldots, \sigma_n \rhd \tau$ permits an interpretation in $C([\![\Gamma]\!], [\![\tau]\!] \otimes [\![\sigma_n]\!] \otimes \cdots \otimes [\![\sigma_1]\!])$ so that $\sigma_1, \ldots, \sigma_n$ will be traced out. We guard the last 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. Note that adding a new syntactic category is straightforward in the tagless final style [6], as demonstrated by the *Defs* type class in Section 2.1.

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

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

4.3 Superposing Law

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 identity discussed in Section 2.1.

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

This, combined with other laws, yields the above-mentioned identity.

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

5 Implementation and Experience

In this section, we report our implementation of the proposed constructs used in the embedded version [22] of FliPpr [21], an invertible pretty-printing system. In FliPpr, users write pretty-printers using tailored pretty-printing combinators [31]

```
rec a ← do a ← b
           putStrLn "Hello"
           pure a
       b ← do pure (pure a) :: IO (IO ())
pure a
```

(a) Code that fails (throws `Exception: cyclic evaluation in fixIO`)

```
rec b ← do pure (pure a) :: IO (IO ())
       a ← do a ← b
              putStrLn "Hello"
              pure a
pure a
```

(b) Code that works (prints `"Hello"`)

Fig. 4: *MonadFix* instances for IO (and $ST\ s$) is error-prone

with a certain restriction, so that it can generate context-free grammars for the printer, with the guarantee that pretty-printed strings must always be correctly parsed.[6] FliPpr uses the MRD builders in two places: as a surface language that users write, and as a target grammar of grammar transformations (Fig. 3).

5.1 Motivation in FliPpr

The motivation of MRD builders in FliPpr is twofold: one is for surface representation and the other is for its internal use. Regarding the surface language, MRD builders are used to define pretty-printers indexed by precedence levels, which are to be converted to mutually recursive definitions indexed by precedence levels; such construction is laborious with a *letrec*-like combinator, which directly translates the **letrec** construct, as we need type-level programming to determine the type of whole bindings to be σ^n for some σ, where n is the maximum precedence. Regarding the internal processing, MRD builders are used to represent target grammars for grammar-to-grammar transformations that happen in the internal processing in FliPpr. An advantage of using MRD builders is that it guarantees the type- and scope-safe treatment of grammars, preventing mistakes.

However, there has been some historical evolution towards it. In the original standalone FliPpr, a pretty-printing function can have static inputs (e.g., precedence level) that are removed via partial evaluation during grammar generation [21]. In the embedded version, such processing of static parameters is implemented as Haskell-level programming with core constructs—that is, Haskell evaluates them away [22]. Originally, the embedded FliPpr marks with a designated primitive the places where recursive calls happen, to be used with an appropriate *MonadFix* instance that gives different names for each occurrence of a mark, which will then be replaced with references that hold recursion bodies. However, we soon found that this approach was error-prone in the internal processing; the behavior of some code depends on the order of bindings in **rec** (see Fig. 4). Thus, we sought a more robust and theoretically-backed way that

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

supports `RecursiveDo` and provides a similar usability with marked recursions. This is where the idea of MRD builders originates.

We note that there is a crucial discrepancy between MRD builders and marked recursions. In the marked recursion, the expansion of the defined names (represented as references) is done dynamically in the processing of syntax trees, and thus, there is no issue with having a pretty-printing function with a static parameter of type *Int*. Thanks to Haskell's lazy semantics, only the relevant parts will be used and evaluated. However, following the *letr* definition that uses *letr1s*-like implementation, having a pretty-printing function indexed by *Int* will yield nonterminals with the number of possible *Int* values. Although we prune unused nonterminals afterwards, *letr* constructs an AST of at least that size, which is clearly infeasible. Fortunately, for precedence, using *Int8*, *Word8*, or custom smaller integer types suffices, where the constructed AST sizes will be feasible. We could avoid the issue by having further intermediate structures, which is left for our future directions.

5.2 Programming in FliPpr

The programming in FliPpr is not that different from the ones presented in Section 2.2. We define EDSL's recursive functions mainly via `RecursiveDo`. For illustration, Fig. 5 shows a snippet of a FliPpr program that uses our combinators. It also uses *share* defined as $share\ e = letr\ \$\ \lambda x \to pure\ (e, x)$, which represents a non-recursive **let**.

It is worth mentioning that there is a limitation intrinsic to `RecursiveDo`, which can only be used in **do** blocks and cannot be defined at the top level. We can use the metaphor that a Haskell function $f\ \sigma_1 \to \cdots \to f\ \sigma_n \to DefM\ f\ (f\ \tau_1, \ldots, f\ \tau_m)$ can be interpreted as an EDSL-level module that imports EDSL objects of types $f\ \sigma_1, \ldots, f\ \sigma_n$ and exports objects of types $f\ \tau_1, \ldots, f\ \tau_m$. However, this requires a shift in programming style. Actually, a recent version of FliPpr is moving towards supporting a surface syntax with implicit recursions, which can be made explicit by observing graph structures in memory [7] using *StableName*. Still, we need a careful treatment for static parameters, as a *StableName* of an application result may differ for each application. We need to perform tabulation of such functions, as performed in the `memoize`[7] package.

5.3 Internal Use in FliPpr

A more crucial use appears in the internals of FliPpr for its grammar-to-grammar transformations (Fig. 3). We note that we do not use the representation for source grammars, i.e., grammars to be traversed; one reason is that HOAS does not support intensional analysis, and the information of the whole mutual bindings is not available for the proposed constructs. Thus, before traversal, we convert ASTs into flat grammars (as used in [3]), namely, $Env\ (RHS\ \Gamma)\ \Gamma$,

[7] https://hackage.haskell.org/package/memoize

```
pprExp :: (Phased s) ⇒ FliPprM s v (In v Exp → F.Exp s v D)
pprExp = F.do
  pprName ← share $ λx → case_ x [unName $ λs → textAs s ident]
  pprInt ← share $ λn → convertInput itoaBij n $ λs → textAs s numbers
  pprBool ← share $ λb → case_ b [unTrue $ text "true", unFalse $ text "false"]
  ...
  rec pExp ← share $ λ(prec :: Word8) (b :: IsRightMost) x →
        if | prec == 4 → case_ x
             [$(pat 'Op) $(pat ' Add) varP varP `br` λe1 e2
                → pprOp pExp b (Fixity AssocL 4) "+" e1 e2
             ,$(pat 'Op) $(pat ' Sub) varP varP `br` λe1 e2
                → pprOp pExp b (Fixity AssocL 4) "-" e1 e2
             , otherwiseP $ pExp (prec + 1) b]
           | prec == 6 → ...
           | prec == 10 → ...
           | prec ⩾ 11 → pSimpleExp b x
           | otherwise → pExp (prec + 1) b x
      pSimpleExp ← share $ λb e0 →
        let br0 = [unLet $ λx e1 e2 → pprLet pExp x e1 e2
                  , unIf $ λe e1 e2 → pprIf pExp e e1 e2
                  , unAbs $ λx e → pprAbs pExp x e]
            br1 = [unVar pprName
                  , unLiteral pprLit
                  , otherwiseP $ parens ∘ pExp 0 (IsRightMost True)]
        in case_ e0 (if isRightMost b then br0 ++ br1 else br1)
      pure $ pExp 0 (IsRightMost True)
```

Fig. 5: Snippet of a FliPpr program: taken from https://github.com/kztk-m/flippre/blob/new-syntax/flippre-examples/SimpleFL.hs

where *RHS* Γ σ denotes right-hand sides of production rules of type σ that can use nonterminals of type Γ. To do so, we first transform HOAS grammars to the equivalent de Bruijn-indexed representation by unembedding [1, 2, 20], and then perform flattening, which is a variant of A-normalization [12], using MRD builders.

Specifically, the proposed combinator is used in processing whitespaces. FliPpr interprets some pretty-printing combinators such as *linebreak*, which may be rendered as the empty string or a newline with indentation depending on width available, as zero-or-more spaces. Since the interpretation happens behind the scenes, FliPpr transforms grammars so that the spaces introduced by the process cannot lead to ambiguity [23]. For example, *linebreak* will be replaced with a grammar that corresponds to a regex `\s*`, but the concatenated `\s*\s*` causes the ambiguity, for example, about how a single space character is produced (by the first `\s*` or the second `\s*`). Of course, it might be rare to concatenate *linebreak*s, but a similar ambiguous grammar is yielded from ... <> *linebreak* <> ..., where

```
-- 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. 6: Use of *letr1* in internal whitespace processing in FliPpr

s <> t allows zero-or-more spaces between s and t in parsing. This transformation is implemented as a three-state transducer, and a transducer can be fused into a grammar; the resulting grammar has nonterminals of the form $N^{q_i q_j}$ where N is a nonterminal in the original grammar, and q_i and q_j are transducer states before and after processing N. However, generating all the combinations is time- and space-consuming, and thus we generate such nonterminals on demand.

Fig. 6 extracts the key use of *letr1* in space processing.[8] The function *procVar* handles nonterminals in a given flat grammar. Here, *DefM* is used together with the *State* monad that passes around memos. This function first checks a memo to see whether a nonterminal x with a given pair of states *q1* and *q2* has already been processed. If an entry is found in the memo, it returns the entry. If there is no corresponding entry in the memo, it first generates a nonterminal corresponding to *q1*, *q2* and x, adds the nonterminal to the entry, and processes the right-hand side of x by *procRHS q1 q2 rhs*. The processing result (r) will be the right-hand side of the generated nonterminal a. The *return* expression in the last line ensures this by *return* $(r, (a, memo'))$, which also returns a as the result of *procVar*. The use of *letr1* is surprisingly simple as only type variables a and g are involved—we need not care directly about the already defined symbols (hidden in *memo*). One may think that *letr1* is a type- and scope-safe version of `gensym`, coupled with the obligation to provide its body.

We note that *letr1* is also used with simple inlining of grammars, where the number of nonterminals in the resulting grammar depends on the concrete structure of the input grammar. We shall not show the code in the paper, as the use of *letr1* is similar.

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

5.4 Performance Note

The nonterminal type *IxN env a* in a flat grammar mentioned in Fig. 6 is indeed a phantom type; it is actually implemented as a *Word* instead of an actual de Bruijn index for performance purposes. With the standard definition of the de Bruijn indices, functions like *lookupMemo* in Fig. 6 took around 90% of the execution time for a fairly large grammar. Another source of the significant slowdown is flattening, which works on de Bruijn-indexed terms. The version used in FliPpr passes around compositions of shifting functions to avoid shifting constructed terms, where such compositions of shifting functions are applied to variables in a given de Bruijn-indexed term. For a fairly large grammar, the shifting amount can be large—it is as large as the number of nonterminals in the resulting flat grammar—and so is the number of compositions. The application of a composed function takes at least time proportional to the number of compositions—even though composed functions are lightweight like a mere increment function $\lambda x \to x + 1$. We need to pay this cost every time a composed function is applied. We observe that our flattening (a variant of A-normalization) tends to compose simple shifting functions such as the one that just unconditionally increments indices to the left. Thus, we use a designated datatype so that we can only remember the total increment amount of such unconditional shifting composed to the left. Although a profiling showed that the flattening after unembedding took around 35% of the execution time, adopting the optimization made the code run around 7 times faster.[9]

Overall, the code runs nearly 100 times faster combined with other optimizations. Although the flattening is an extreme case—processing of first-order expressions such as e `<*>` e' requires shifting as each processing result comes with lifted bindings—we would expect that a similar implementation technique would be useful for implementing the conversion in Section 3.2. Notice that the conversion of **push** $e\ d$ requires shifting (weakening) to put e typed under the context Γ into the work list, which is typed under the context Γ, Δ.

6 Discussion and Related Work

We first discuss the representation of mutually recursive definitions. As a benchmark representation, we considered the straightforward HOAS representation of **letrec** $\overline{x = e}$ **in** e', given as

$$\begin{aligned} &\mathit{letrec} :: \mathit{Env}\ \mathit{Proxy}\ \Delta \\ &\qquad \to (\mathit{Env}\ \mathit{Exp}\ \Delta \to \mathit{Env}\ \mathit{Exp}\ \Delta) \to (\mathit{Env}\ \mathit{Exp}\ \Delta \to \mathit{Exp}\ a) \to \mathit{Exp}\ a \end{aligned}$$

and its tupled variant:

$$\mathit{letrec}' :: \mathit{Env}\ \mathit{Proxy}\ \Delta \to (\mathit{Env}\ \mathit{Exp}\ \Delta \to (\mathit{Env}\ \mathit{Exp}\ \Delta, \mathit{Exp}\ a)) \to \mathit{Exp}\ a$$

A similar representation is used by Oliveira and Löh [29]. Notice that this *letrec* takes the shape information as the first argument, which is crucial for some

[9] This compares single runs in the profiling mode.

interpretations such as those that use initial values to compute a fixed point. Otherwise, since we can know the shape of Δ only by pattern matching $Env\ f\ \Delta$, what we can do would be limited to tying the knot. That is, EDSL recursion will be implemented by Haskell's recursion. For such cases, there is no strong reason to have EDSL-level mutually recursive definitions. Kiselyov shows that we can avoid passing the shape information with the following variation [17].

$$letrec'' :: Env\ ((Compose\ (\to)\ (Env\ Exp\ \Delta))\ Exp)\ \Delta \to (Env\ Exp\ \Delta \to Exp\ a) \to Exp\ a$$

Observe that we have $(Compose\ ((\to)\ (Env\ Exp\ \Delta))\ Exp)\ x \simeq Env\ Exp\ \Delta \to Exp\ x$. In this representation, the shape information can be obtained by applying $fmap\ (const\ Proxy)$ to its first argument. An inconvenience in this variant is that the size of the recursive bindings will be quadratic in the number of recursive bindings.

Baars et al. [3] use a type like $Env\ (RHS\ \Delta)\ \Delta$ for recursive grammars without start symbols, where $RHS\ \Delta\ a$ is a type for right-hand sides of production rules of type a that may refer to nonterminals of type Δ. That is, $RHS\ \Delta\ a$ represents de Bruijn-indexed terms. Thus, writing and generating programs is laborious in this representation, while some interpretations are known to be difficult when we use the HOAS representation [1,2,20] instead.

Devriese and Piessens [10] and Brink et al. [4] use the fixed-point combinator like the following *kfix*.

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

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. For example, for rose trees **data** $RTree = RNode\ [RTree]$, the size counting functions can be defined as:

```
kfix $ λh → λcase
  NodeK → lam $ λx → case_ x [unRNode $ λts → add one (h ListK ts)]
  ListK → lam $ λx → case_ x
    [unNil $ zero,
     unCons $ λy ys → add (h NodeK y) (h ListK ys)
```

assuming some appropriate EDSL constructs such as *add*, where *NodeK* and *ListK* are the constructors of the following GADT.

```
data Key a where
  NodeK :: Key (RTree → Int)
  ListK :: Key ([RTree] → Int)
```

That is, instead of tuples $(Env\ Exp\ \Delta)$, this representation uses branches. This representation requires us to prepare appropriate data types in advance, and the interpretation needs to know the constructors of the *Key* type. Thus, it would be inconvenient for an EDSL interface.

Benedikt et al. [28] use the combinator $def :: Def\ a\ f \Rightarrow (a \to (a, f\ b)) \to f\ b$ for recursive definitions. Although its type resembles our $letr :: LetRecArg\ m\ t \Rightarrow (t \to m\ (t, r)) \to m\ r$, there is a critical difference. Our *letr* is a *derived* instance while their *def* is a core primitive. Being a core primitive, the type of *def* is too general without restricting a; as a ranges over the host's types, the only possible interpretation would be the tying-the-knot, i.e., $def\ f = \mathbf{let}\ (a, r) = f\ a\ \mathbf{in}\ r$. Thus, we need appropriate restriction of a to have non-trivial interpretations, whereas they address this in an ad-hoc manner for each interpretation (e.g., printing). Another crucial difference is that the return type of *def* is restricted to a single EDSL expression type $f\ b$, while *letr*'s return is an arbitrary Haskell type r. As we demonstrated in Section 2.2, this generality is crucial for defining mutually recursive definitions one-by-one.

The EDSL `accelerate` adopts the reification of implicit sharing into EDSL's **let** [25], relying on a specific behavior of GHC that cyclic first-order data are evaluated into cyclic structures in memory. For example, by $\mathbf{let}\ ones = 1 : ones\ \mathbf{in}\ ones$, after the evaluation, we will get a cons cell whose cdr part points to itself. GHC provides a way to compare such pointers via *StableName*, and thus, by leveraging *StableName*, one can detect graph structures in memory to obtain **letrec**s in an EDSL. Although the original method targets non-recursive **let** instead of **letrec**, the method can be extended to recursive **let**. An advantage of this approach is the extreme simplicity in EDSL programming: especially, it allows top-level recursive definitions. While this representation is good for manual programming by users, it is error-prone if we programmatically generate EDSL expressions.

Unembedding [1] provides an interconversion between (parametric/tagless-final) HOAS representation and de Bruijn-indexed representation of simply-typed ASTs, which is provably correct based on Kripke parametricity. An advantage of the conversion to de Bruijn-indexed representation is its support for intensional analysis of ASTs, for example, for optimization [2], while keeping the programmability coming from the HOAS representation. Another advantage is its support for interpretation of open terms [20], which is crucial for incremental computation [5, 14] and bidirectional transformations [13, 24]. Section 3.1 highlights another use case: supporting existentially-quantified semantic domains, for which the HOAS representation changes the timing of choice of existentials.

7 Conclusion

In this paper, we proposed mutually recursive definition builder constructs, whose usefulness lies especially in EDSL program generation. The idea of the constructs comes from 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 reported our experience using the proposed constructs in the invertible pretty-printing system FliPpr, especially for their utility in grammar generation.

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, 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, http://doi.acm.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, https://doi.org/10.1016/j.entcs.2010.08.031
4. 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, https://doi.org/10.1007/978-3-642-13321-3_6
5. 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, https://doi.org/10.1145/2594291.2594304
6. 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, https://doi.org/10.1017/S0956796809007205
7. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Carro, M., Reppy, J.H. (eds.) Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011. pp. 3–14. ACM (2011). https://doi.org/10.1145/1926354.1926358, http://doi.acm.org/10.1145/1926354.1926358
8. 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, http://doi.acm.org/10.1145/1411204.1411226
9. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170, http://dx.doi.org/10.2307/2266170
10. 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, https://doi.org/10.1017/S0956796812000226
11. 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, https://doi.org/10.1145/237721.237792

12. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993. pp. 237–247. ACM (1993). https://doi.org/10.1145/155090.155113, https://doi.org/10.1145/155090.155113
13. 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)
14. 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, https://doi.org/10.1007/978-3-030-17184-1_20
15. 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, http://dx.doi.org/10.1007/BF00264598
16. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society **119**(3), 447–468 (Apr 1996)
17. 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
18. 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, https://doi.org/10.1145/1017472.1017488
19. Kock, A.: Continuous yoneda representations of a small category. Preprint (1966), available on http://tildeweb.au.dk/au76680/CYRSC.pdf
20. 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, https://doi.org/10.1145/3607830
21. 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, https://doi.org/10.1007/978-3-642-37036-6_6
22. 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, https://doi.org/10.1145/3242744.3242758
23. 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, https://doi.org/10.1007/s00354-018-0033-7
24. 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, https://doi.org/10.1007/978-3-319-89884-1_2

25. 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, https://doi.org/10.1145/2500365.2500595
26. 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)
27. 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, http://doi.acm.org/10.1145/53990.54010
28. 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, https://doi.org/10.1145/3756907.3756923
29. 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, https://doi.org/10.1145/2426890.2426909
30. 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, https://doi.org/10.1007/978-3-540-70594-9_20
31. Wadler, P.: A prettier printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, chap. 11. Palgrave Macmillan (2003)