Handling Recursion in Generic Programming Using Closed Type Families

Anna Bolotina[1] and Artem Pelenitsyn[2]

[1] Southern Federal University, Russia
bolotina@sfedu.ru

[2] Czech Technical University in Prague, Czech Republic
pelenart@fit.cvut.cz

June 12, 2018

The 19th International Symposium on Trends in Functional Programming (2018)
TFP 2018

Contents

Handling Recursion in Generic Programming (GP)

Many generic functions consider information on the recursion points when traversing the structure of datatypes.

Examples: *maps* [5] and *folds* [7]. More advanced one: a *zipper* [4].

How to obtain that information?

1. **Solution I:** A GP framework should be explicit about the recursion encoding in the datatype representation.

   Examples: The libraries regular [8], multirec [9] use fixed points to capture recursion.

Downside

This may complicate the whole GP framework significantly.

2. **Solution II:** Using global or local overlapping instances.

Downside

This complicates the semantics of code, makes that unstable.

Case Study: The *True Sums of Products* (SOP) Framework

The SOP [1] approach to datatype-generic programming is implemented in the generics-sop library.

- This does not reflect recursive positions in the generic representation of a datatype.
- Datatypes are expressed as *n*-ary sums of *n*-ary products of types.

An *n*-ary product example (heterogeneous list)

```
I 5 :* I True :* I 'x' :* Nil :: NP I '[Int, Bool, Char]
```

An *n*-ary sum example (choice)

```
S (S (Z (I 5))) :: NS I '[Char, Bool, Int, Bool]
```

Example of a datatype representation

```
data Tree a
     = Leaf a
     | Node (Tree a) (Tree a)
```

```
type RepTree a = NS (NP I) (
          '[ '[a]
           , '[Tree a, Tree a]
           ])
```

Example: The Generic Function subterms

The function subterms takes a term and obtains a list of all its immediate subterms that are of the same type as the given term.

Implementation of subterms using the SOP view

```
subterms :: Generic a => a -> [a]
subterms t = subtermsNS (unSOP $ from t)

subtermsNS :: NS (NP I) xss -> [a]
subtermsNS (S ns) = subtermsNS ns
subtermsNS (Z np) = subtermsNP np

subtermsNP :: ∀a xs. NP I xs -> [a]
subtermsNP p (I y :* ys)
    | typeOf @a y = witnessEq y : subtermsNP ys
    | otherwise   = subtermsNP ys
subtermsNP _ Nil  = []
```

(Bad) Solution with Overlapping Instances

We need a way to check type equality and witness the coercion between equal types.

Implementation of `subtermsNP` using overlapping instances

```
class Subterms a (xs :: [*]) where
    subtermsNP :: NP I xs -> [a]

instance Subterms a xs => Subterms a (x ': xs) where
    subtermsNP (_   :* xs) = subtermsNP xs
instance {-# OVERLAPS #-} Subterms a xs
        => Subterms a (a ': xs) where
    subtermsNP (I x :* xs) = x : subtermsNP xs
instance Subterms a '[] where
    subtermsNP _ = []
```

Although the approach works, we feel this unsatisfactory, and go to a revised solution free of overlap.

Proof for Type-Level Equality

Closed type families [2] were introduced in Haskell to solve the overlap problem.

Type equality

```
type family Equal a x :: Bool where
    Equal a a = 'True
    Equal a x = 'False
```

Witnessing the coercion

```
class Proof (eq :: Bool) (a :: *) (b :: *) where
    witnessEq :: b -> Maybe a

instance Proof 'False a b where
    witnessEq = Nothing
instance Proof 'True a a where
    witnessEq = Just
```

Solution to subtermsNP revised

Abbreviation for Proof

```
class    Proof (Equal a b) a b => ProofEq a b
instance Proof (Equal a b) a b => ProofEq a b
```

All applies a particular constraint to each member of a list of types.

Implementation of subtermsNP using Proof of type equality

```
subtermsNP :: ∀a xs. All (ProofEq a) xs => NP I xs -> [a]
subtermsNP (I (y :: x) :* ys)
    = case witnessEq @(Equal a x) y of
          Just t  -> t : subtermsNP ys
          Nothing -> subtermsNP ys
subtermsNP Nil = []
```

Generic Zipper Interface

The Zipper [3] represents a current location in a datatype structure, storing a tree node, a *focus*, along with its context.

Movement functions

```
goUp    :: Loc a fam c -> Maybe (Loc a fam c)
goDown  :: Loc a fam c -> Maybe (Loc a fam c)
goLeft  :: Loc a fam c -> Maybe (Loc a fam c)
goRight :: Loc a fam c -> Maybe (Loc a fam c)
```

Starting navigation

```
enter   :: ∀fam c a. (Generic a, In a fam, Zipper a fam c)
        => a -> Loc a fam c
```

Ending navigation

```
leave   :: Loc a fam c -> a
```

Updating

```
update  :: (∀b. c b => b -> b) -> Loc a fam c -> Loc a fam c
```

Usage I

Example of mutually recursive datatypes

```
data RoseTree a = RTree a (Forest a)

data Forest   a = Empty | Forest (RoseTree a) (Forest a)
```

Class for updating trees

```
class UpdateTree a b where
    replaceBy :: RoseTree a -> b -> b
    replaceBy   = id

instance UpdateTree a (RoseTree a) where
    replaceBy t = t
instance UpdateTree a (Forest a)
```

Usage II

Chaining moves and edits

```
(>>>) :: (a -> b) -> (b -> c) -> (a -> c)
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
```

Example of usage

```
type TreeFam a = '[RoseTree a, Forest a]

*Main> let forest
          = Forest (RTree 'a' $ Forest (RTree 'b' Empty) Empty)
                   (Forest (RTree 'x' Empty) Empty)

*Main> let t = RoseTree 'c' Empty

*Main> enter @(TreeFam Char) @(UpdateTree Char)
          >>> goDown >=> goRight >=> goDown
          >=> update (replaceBy t)
          >>> leave >>> return $ forest
```

```
Forest (RTree 'a' $ Forest (RTree 'b' Empty) Empty)
       (Forest (RTree 'c' Empty) Empty)
```

Datatype of Locations

Datatype of locations

```
data Loc (r :: *) (fam :: [*]) (c :: * -> Constraint) where
    Loc :: Focus r a fam c
        -> Contexts r a fam c
        -> Loc r fam c
```

Meanings of the type parameters

- `r` — the root type of the tree;
- `fam` — the list of types of nodes to visit (family);
- `c` — constraint imposing restrictions on the types in the list;
- `a` — a type of the focus' parent.

Focus

Focus

```
data Focus (r :: *) (a :: *) (fam :: [*])
           (c :: * -> Constraint) where
    Focus :: (Generic b, In b fam, ZipperI r a b fam c)
          => b -> Focus r a fam c
```

```
type In a fam = InFam a fam ~ 'True
```

Proof for Focus

This proof generalizes the proof of type equality.

```
class ProofFocus (inFam :: Bool) (r :: *) (a :: *) (b :: *)
                 (fam :: [*]) (c :: * -> Constraint) where
    witness :: b -> Maybe (Focus r a fam c)

instance ProofFocus 'False r a b fam c where
    witness = Nothing
instance (Generic b, In b fam, ZipperI r a b fam c)
        => ProofFocus 'True r a b fam c where
    witness = Just . Focus

class ProofFocus (InFam b fam) r a b fam c
        => ProofIn r a b fam c
instance ProofFocus (InFam b fam) r a b fam c
        => ProofIn r a b fam c
```

Contexts

- The context can be expressed as a stack, called `Contexts`;
- Each frame, `Context`, corresponds to the particular node with a hole.

Datatype of contexts

```
data Contexts (r :: *) (a :: *) (fam :: [*])
              (c :: * -> Constraint) where
    CNil :: Contexts a a fam c
    Ctxs :: (Generic a, In a fam, ZipperI r x a fam c)
         => Context fam a -> Contexts r x fam c
         -> Contexts r a fam c
```

Type-level Differentiation

"The derivative of a regular type is its type of one-hole contexts." (McBride) [6]

Defining type-level algebraic operations

- Sum of products (SOP) $+$ $(.+)$ — appends two type-level lists of lists;
- SOP-by-product $\times$ $(.*)$ — appends the list to the head of each inner product of the sum.

Context Frame

Differentiation of a product of type

```
type family DiffProd (fam :: [*]) (xs :: [*]) :: [[*]] where
    DiffProd fam '[]        = '[]
    DiffProd fam '[x]       = If (InFam x fam) '[ '[]] '[]
    DiffProd fam (x ': xs)
        = xs .* DiffProd fam '[x] .++ '[x] .* DiffProd fam xs
```

Computation of the context type

```
type family ToContext (fam :: [*]) (code :: [[*]]) :: [[*]] where
    ToContext fam '[] = '[]
    ToContext fam (xs ': xss)
        = DiffProd fam xs .++ ToContext fam xss
```

```
newtype Context fam a = Ctx {ctx :: SOP I (CtxCode fam a)}
```

Function goDown

Definition of goDown

```
goDown :: Loc a fam c -> Maybe (Loc a fam c)
goDown (Loc (Focus t) cs)
    = case toFirst t of
          Just t' -> Just $ Loc t' (Ctxs (toFirstCtx t) cs)
          _       -> Nothing
```

This uses two auxiliary functions:

- `toFirst` — analyzes the focal subtree's representation to find its first immediate child;
- `toFirstCtx` — computes its respective context.

Implementation of toFirst

```
toFirst :: ∀fam c r a. (Generic a, ToFirst r a fam c)
        => a -> Maybe (Focus r a fam c)
toFirst t = appToNP @AllProof toFirstNP $ unSOP $ from t
```

Proof

```
class    All (ProofIn r a fam c) xs => AllProof r a fam c xs
instance All (ProofIn r a fam c) xs => AllProof r a fam c xs

type ToFirst r a fam c = All (AllProof r a fam c) (Code a)
```

Processing products

```
toFirstNP :: ∀fam c r a xs. All (ProofIn r a fam c) xs
          => NP I xs -> Maybe (Focus r a fam c)
toFirstNP (I (x :: b) :* xs)
    = witness @(InFam b fam) x `mplus` toFirstNP xs
toFirstNP Nil = Nothing
```

The full implementation of the zipper interface is available at https://github.com/Maryann13/Zipper.

References

[1] E. De Vries and A. Löh. True sums of products. *WGP* '14.

[2] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. *POPL* '14.

[3] G. Huet. The zipper. *JFP*, 1997.

[4] A. Löh and J. P. Magalhães. Generic programming with indexed functors. *WGP* '11.

[5] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. *Haskell* '10.

[6] C. McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, 2001.

[7] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. *FPLCA* '91. Berlin Heidelberg.

[8] T. Van Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. *WGP* '08.

[9] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. *ICFP* '09.