Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Generic programming with the multirec library

Andres Löh

joint work with Alexey Rodriguez, Stefan Holdermans, Johan Jeuring

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Web pages: `http://www.cs.uu.nl/wiki/Center`

May 15, 2009

Why another generic programming library?

Overview

This talk:

- Generic programming with fixed points of datatypes
- Generalizing to families of datatypes
- Examples

Introduction:
Generic programming, PolyP style, in Haskell

Idea

- Express (regular) functors as fixed points of functors (user).
- Use a limited set of combinators to build functors (library).
- Express the equivalence using a pair of conversion functions (user).
- Define functions (and datatypes) on the structure of functors (library).
- Enjoy generic functions on all the represented datatypes (user).

Running example: expressions

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul
```

Running example: expressions

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul
```

Corresponding functor:

```
data PFExpr' r = PFOne
               | PFNeg r
               | PFBin  r Op r
```

Running example: expressions

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul
```

Corresponding functor:

```
data PFExpr' r = PFOne
               | PFNeg r
               | PFBin  r Op r
```

Or in terms of the building blocks:

```
type PFExpr  =  U
             :+: I
             :+: I :×: K Op :×: I
```

Regular functors

Combinators for regular functors:

```
data I         r = I r
data K a       r = K a
data U         r = U
data (f :+: g) r = L (f r) | R (g r)
data (f :×: g) r = f r :×: g r
```

Converting between datatype and functor

```
fromExpr :: Expr → PFExpr Expr
fromExpr One            = L U
fromExpr (Neg e)        = R (L (I e))
fromExpr (Bin e1 o e2) = R (R (I e1 :×: K o :×: I e2))

toExpr :: PFExpr Expr → Expr
toExpr = ...
```

Functors are Haskell functors

class Functor f **where**
  $\mathsf{map} :: \forall a\ b.(a \to b) \to (f\ a \to f\ b)$

Functors are Haskell functors – generically

```
class Functor f where
  map :: ∀a b.(a → b) → (f a → f b)

instance Functor I       where
  map φ (I r) = I (φ r)
instance Functor (K a) where
  map φ (K a) = K a
instance Functor U       where
  map φ U = U
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  map φ (L f)  = L (map φ f)
  map φ (R g) = R (map φ g)
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  map φ (f :×: g) = map φ f :×: map φ g
```

Capturing the generic description

```
type family PF a :: * → *
class (Functor (PF a)) ⇒ Regular a where
   from :: a → PF a a
   to   :: PF a a → a
```

Capturing the generic description

```
type family PF a :: * → *
class (Functor (PF a)) ⇒ Regular a where
  from :: a → PF a a
  to   :: PF a a → a
```

Expressions:

```
type instance PF Expr = PFExpr
instance Regular Expr where
  from = fromExpr
  to   = toExpr
```

Fixed points

```
data Fix f = In {out :: f (Fix f)}

in' :: Regular a ⇒ a → Fix (PF a)
in' = In ∘ map in' ∘ from

out' :: Regular a ⇒ Fix (PF a) → a
out' = to ∘ map out' ∘ out
```

Fixed points

```
data Fix f = In {out :: f (Fix f)}

in' :: Regular a ⇒ a → Fix (PF a)
in' = In ∘ map in' ∘ from

out' :: Regular a ⇒ Fix (PF a) → a
out' = to ∘ map out' ∘ out
```

Fold, unfold and compos, generically:

```
fold :: Regular a ⇒ (PF a r → r) → (a → r)
fold φ = φ ∘ map (fold φ) ∘ from

unfold :: Regular a ⇒ (r → PF a r) → (r → a)
unfold φ = to ∘ map (unfold φ) ∘ φ

compos :: Regular a ⇒ (a → a) → (a → a)
compos φ = to ∘ map φ ∘ from
```

Intermediate Summary

The library provides

- functor combinators K, U, I, :+:, :×:,
- type family PF and class Regular,
- inductively defined generic functions such as map,
- derived generic functions such as fold or compos.

Intermediate Summary

The library provides

- functor combinators K, U, I, :+:, :×:,
- type family PF and class Regular,
- inductively defined generic functions such as map,
- derived generic functions such as fold or compos.

To add a new function

- give an inductive or derived definition.

Intermediate Summary

The library provides

- functor combinators K, U, I, :+:, :×:,
- type family PF and class Regular,
- inductively defined generic functions such as map,
- derived generic functions such as fold or compos.

To add a new function

- give an inductive or derived definition.

To use on a new datatype

- define PF and Regular instances (can be done using Template Haskell).

Generalizing to families of mutually recursive datatypes

Fixed points

$$\mathsf{Fix}_1 : (* \to *) \to *$$
$$\mathsf{Fix}_1\ f = f\ (\mathsf{Fix}_1\ f)$$

$$\mathsf{Fix}_{2,0} : (* \to * \to *) \to (* \to * \to *) \to *$$

$$\mathsf{Fix}_{2,1} : (* \to * \to *) \to (* \to * \to *) \to *$$

$$\begin{aligned} \mathsf{Fix}_{2,0}\ f_0\ f_1 &= f_0\ (\mathsf{Fix}_{2,0}\ f_0\ f_1)\ (\mathsf{Fix}_{2,1}\ f_0\ f_1) \\ \mathsf{Fix}_{2,1}\ f_0\ f_1 &= f_1\ (\mathsf{Fix}_{2,0}\ f_0\ f_1)\ (\mathsf{Fix}_{2,1}\ f_0\ f_1) \end{aligned}$$

$$\mathsf{Fix}_n : ?$$

Fixed points

$$\mathsf{Fix}_1 : (* \to *) \to *$$
$$\mathsf{Fix}_1\ f = f\ (\mathsf{Fix}_1\ f)$$

$$\mathsf{Fix}_{2,0} : (* \times * \to *) \times (* \times * \to *) \to * \qquad \mathsf{Fix}_{2,1} : (* \times * \to *) \times (* \times * \to *) \to *$$

$$\mathsf{Fix}_{2,0}\ (f_0, f_1) = f_0\ (\mathsf{Fix}_{2,0}\ (f_0, f_1))\ (\mathsf{Fix}_{2,1}\ (f_0, f_1))$$
$$\mathsf{Fix}_{2,1}\ (f_0, f_1) = f_1\ (\mathsf{Fix}_{2,0}\ (f_0, f_1))\ (\mathsf{Fix}_{2,1}\ (f_0, f_1))$$

$$\mathsf{Fix}_n : ?$$

Fixed points

$$\mathrm{Fix}_1 \quad : (* \to *) \to *$$

$$\mathrm{Fix}_1\ f = f\ (\mathrm{Fix}_1\ f)$$

$$\mathrm{Fix}_2 \quad : (* \times * \to *) \times (* \times * \to *) \to * \times *$$

$$\begin{aligned}\mathrm{Fix}_2\ (f_0, f_1) = (&f_0\ (\mathrm{fst}\ (\mathrm{Fix}_2\ (f_0, f_1)))\ (\mathrm{snd}\ (\mathrm{Fix}_2\ (f_0, f_1))),\\ &f_1\ (\mathrm{fst}\ (\mathrm{Fix}_2\ (f_0, f_1)))\ (\mathrm{snd}\ (\mathrm{Fix}_2\ (f_0, f_1))))\end{aligned}$$

$$\mathrm{Fix}_n \quad : ?$$

Fixed points

$$\mathsf{Fix}_1 \quad : (* \to *) \to *$$

$$\mathsf{Fix}_1\ \mathsf{f} = \mathsf{f}\ (\mathsf{Fix}_1\ \mathsf{f})$$

$$\mathsf{Fix}_2 \quad : \quad (*^2 \to *)^2 \to *^2$$

$$\begin{aligned}\mathsf{Fix}_2\ (\mathsf{f}_0, \mathsf{f}_1) = (&\mathsf{f}_0\ (\mathsf{fst}\ (\mathsf{Fix}_2\ (\mathsf{f}_0, \mathsf{f}_1)))\ (\mathsf{snd}\ (\mathsf{Fix}_2\ (\mathsf{f}_0, \mathsf{f}_1))), \\ &\mathsf{f}_1\ (\mathsf{fst}\ (\mathsf{Fix}_2\ (\mathsf{f}_0, \mathsf{f}_1)))\ (\mathsf{snd}\ (\mathsf{Fix}_2\ (\mathsf{f}_0, \mathsf{f}_1))))\end{aligned}$$

$$\mathsf{Fix}_n \quad : ?$$

Fixed points

$$\mathsf{Fix}_1 \quad : (* \to *) \to$$
$$*$$
$$\mathsf{Fix}_1\ \mathsf{f} = \mathsf{f}\ (\mathsf{Fix}_1\ \mathsf{f})$$

$$\mathsf{Fix}_2 \quad : \quad (2 \to (2 \to *) \to *) \quad \to$$
$$(2 \to *)$$
$$\mathsf{Fix}_2 \quad \mathsf{f}\ \mathsf{n} \quad = \mathsf{f}\ \mathsf{n}\ (\mathsf{Fix}\ \mathsf{f})$$

$$\mathsf{Fix}_\mathsf{n} \quad : \textbf{?}$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Fixed points

$$\mathsf{Fix}_1 \quad : (* \rightarrow *) \rightarrow$$
$$*$$
$$\mathsf{Fix}_1\ \mathsf{f} = \mathsf{f}\ (\mathsf{Fix}_1\ \mathsf{f})$$

$$\mathsf{Fix}_2 \quad : \quad ((2 \rightarrow *) \rightarrow (2 \rightarrow *)) \rightarrow$$
$$(2 \rightarrow *)$$
$$\mathsf{Fix}_2 \quad \mathsf{f}\ \mathsf{n} \quad = \mathsf{f}\ (\mathsf{Fix}\ \mathsf{f})\ \mathsf{n}$$

$$\mathsf{Fix}_\mathsf{n} \quad : \quad ((\mathsf{n} \rightarrow *) \rightarrow (\mathsf{n} \rightarrow *)) \rightarrow (\mathsf{n} \rightarrow *)$$

Generalizing fixed points

Before:

$$\mathsf{Fix} : (* \to *) \to *$$

Now:

$$\mathsf{Fix}_{\mathsf{n}} : ((\mathsf{n} \to *) \to (\mathsf{n} \to *)) \to (\mathsf{n} \to *)$$

How to encode n in Haskell?

Extended running example

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul | Infix Expr
```

Extended running example

```
data Expr = One
          | Neg Expr
          | Bin Expr Op Expr
data Op   = Add | Mul | Infix Expr
```

Corresponding index GADT:

$$\mathbf{data}\ \mathsf{ExprOp} :: *_{\mathsf{ExprOp}} \to *\ \mathbf{where}$$
$$\quad \mathsf{Expr} :: \mathsf{ExprOp\ Expr}$$
$$\quad \mathsf{Op} :: \mathsf{ExprOp\ Op}$$

We can use $*_{\mathsf{ExprOp}}$ instead of 2.

Functor

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul | Infix Expr
```

Functor

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul | Infix Expr
```

Corresponding functor:

```
data PFExprOp' :: (*ExprOp → *) → *ExprOp → *where
  PFOne :: PFExprOp' r Expr
  PFNeg :: r Expr → PFExprOp' r Expr
  PFBin  :: r Expr → r Op → r Expr → PFExprOp' r Expr
  PFAdd :: PFExprOp' r Op
  PFMul :: PFExprOp' r Op
  PFInfix :: r Expr → PFExprOp' r Op
```

The plan now

Generalize …

- …the functor combinators U, K, I, :+:, :×:.
- …the type family PF and the class Regular for the conversion functions from and to.
- …the class Functor.

Indexed functor combinators

Most combinators are simply lifted from $* \to *$ to $(*_\varphi \to *) \to (*_\varphi \to *)$:

$$
\begin{array}{lllll}
\textbf{data} & \mathsf{K\ a} & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{K\ a} \\
\textbf{data} & \mathsf{U} & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{U} \\
\textbf{data} & (\mathsf{f} :+: \mathsf{g}) & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{L\ (f\ r\ ix)} \mid \mathsf{R\ (g\ r\ ix)} \\
\textbf{data} & (\mathsf{f} :\times: \mathsf{g}) & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{f\ r\ ix} :\times: \mathsf{g\ r\ ix}
\end{array}
$$

Indexed functor combinators

Most combinators are simply lifted from $* \to *$ to $(*_\varphi \to *) \to (*_\varphi \to *)$:

```
data K a       (r :: *φ → *) ix = K a
data U         (r :: *φ → *) ix = U
data (f :+: g) (r :: *φ → *) ix = L (f r ix) | R (g r ix)
data (f :×: g) (r :: *φ → *) ix = f r ix :×: g r ix
```

Recursive positions change the index:

```
data I xi r ix = I (r xi)
```

Indexed functor combinators

Most combinators are simply lifted from $* \to *$ to $(*_\varphi \to *) \to (*_\varphi \to *)$:

$$
\begin{array}{llll}
\textbf{data}\ \mathsf{K\ a} & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{K\ a} \\
\textbf{data}\ \mathsf{U} & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{U} \\
\textbf{data}\ (\mathsf{f} :+: \mathsf{g}) & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{L\ (f\ r\ ix)} \mid \mathsf{R\ (g\ r\ ix)} \\
\textbf{data}\ (\mathsf{f} :\times: \mathsf{g}) & (\mathsf{r} :: *_\varphi \to *)\ \mathsf{ix} & = & \mathsf{f\ r\ ix} :\times: \mathsf{g\ r\ ix}
\end{array}
$$

Recursive positions change the index:

$$\textbf{data}\ \mathsf{I\ xi\ r\ ix} = \mathsf{I\ (r\ xi)}$$

Tags filter according to the selected index:

$$
\begin{array}{l}
\textbf{data}\ (\mathsf{f} :\triangleright: \mathsf{xi})\ \mathsf{r\ ix}\ \textbf{where} \\
\quad \mathsf{Tag} :: \mathsf{f\ r\ xi} \to (\mathsf{f} :\triangleright: \mathsf{xi})\ \mathsf{r\ xi}
\end{array}
$$

Expressing the example family

```
data Expr = One
          | Neg Expr
          | Bin  Expr Op Expr
data Op   = Add | Mul | Infix Expr
```

```
type PFExprOp = (     U
                :+: I Expr
                :+: I Expr :×: I Op :×: I Expr
                ) :▷: Expr
                :+:
                (     U
                :+: U
                :+: I Expr
                ) :▷: Op
```

Generalizing PF and Regular

Before:

```
type family PF a :: ∗ → ∗
class (Functor (PF a)) ⇒ Regular a where
  from :: a → PF a a
  to   :: PF a a → a
```

Generalizing PF and Regular

Before:

$$\textbf{type}\ \text{family PF a} :: * \rightarrow *$$

$$\textbf{class}\ (\text{Functor (PF a)}) \Rightarrow \text{Regular a}\ \textbf{where}$$
$$\quad \text{from} :: \text{a} \rightarrow \text{PF a a}$$
$$\quad \text{to} \quad :: \text{PF a a} \rightarrow \text{a}$$

Now:

$$\textbf{type}\ \text{family PF}\ \varphi :: (*_\varphi \rightarrow *) \rightarrow (*_\varphi \rightarrow *)$$

$$\textbf{class}\ (\text{HFunctor}\ \varphi\ (\text{PF}\ \varphi)) \Rightarrow \text{Fam}\ \varphi\ \textbf{where}$$
$$\quad \text{from} :: \varphi\ \text{ix} \rightarrow \text{ix} \rightarrow \text{PF}\ \varphi\ \text{I}_*\ \text{ix}$$
$$\quad \text{to} \quad :: \varphi\ \text{ix} \rightarrow \text{PF}\ \varphi\ \text{I}_*\ \text{ix} \rightarrow \text{ix}$$

$$\textbf{newtype}\ \text{I}_*\ \text{x} = \text{I}_*\ \text{x}$$

The φ ix serves as a proof that ix is in φ.

Instances for the example family

```
type instance PF ExprOp = PFExprOp
instance Fam ExprOp where
  from = fromExprOp
  to   = toExprOp
```

The conversions are straight-forward and uninteresting.

All this can be generated using Template Haskell.

Generalizing Functor

```
class HFunctor φ f where
  hmap :: ∀r r' ix.
            (∀ix.φ ix → r ix → r' ix) →
            f r ix → f r' ix
```

Functor instances

Most instances are uninteresting:

```
instance HFunctor φ U where
  hmap f U = U
instance HFunctor φ (K a) where
  hmap f (K x) = K x
instance (HFunctor φ f, HFunctor φ g) ⇒
         HFunctor φ (f :+: g) where
  hmap f (L x) = L (hmap f x)
  hmap f (R x) = R (hmap f x)
instance (HFunctor φ f, HFunctor φ g) ⇒
         HFunctor φ (f :×: g) where
  hmap f (x :×: y) = hmap f x :×: hmap f y
```

Functor for recursive positions and tags

Tags are simple as well. Filtering does not interact with mapping:

$$\textbf{instance}\ \mathsf{HFunctor}\ \varphi\ \mathsf{f} \Rightarrow \mathsf{HFunctor}\ \varphi\ (\mathsf{f} :\triangleright: \mathsf{ix})\ \textbf{where}$$
$$\quad \mathsf{hmap}\ \mathsf{f}\ (\mathsf{Tag}\ \mathsf{x}) = \mathsf{Tag}\ (\mathsf{hmap}\ \mathsf{f}\ \mathsf{x})$$

Functor for recursive positions and tags

Tags are simple as well. Filtering does not interact with mapping:

```
instance HFunctor φ f ⇒ HFunctor φ (f :▷: ix) where
  hmap f (Tag x) = Tag (hmap f x)
```

For recursive calls, we have to change the index:

```
instance (El φ xi) ⇒ HFunctor φ (I xi) where
  hmap f (I x) = I (f proof x)
```

The constraint El φ xi is a class-version of a φ ix:

```
class El φ ix where
  proof :: φ ix
instance El ExprOp Expr where proof = Expr
instance El ExprOp Op   where proof = Op
```

Fixed points and fold

```
data HFix f ix = HIn {hout :: f (HFix f) ix}
```

Fold, unfold and compos, generically:

$$
\begin{aligned}
&\mathsf{fold} :: \forall \varphi\ \mathsf{r}\ \mathsf{ix}.\mathsf{Fam}\ \varphi \Rightarrow \\
&\qquad (\forall \mathsf{ix}.\varphi\ \mathsf{ix} \to \mathsf{PF}\ \varphi\ \mathsf{r}\ \mathsf{ix} \to \mathsf{r}\ \mathsf{ix}) \to (\varphi\ \mathsf{ix} \to \mathsf{ix} \to \mathsf{r}\ \mathsf{ix}) \\
&\mathsf{fold}\ \varphi\ \mathsf{p} = \varphi\ \mathsf{p} \circ \mathsf{hmap}\ (\lambda \mathsf{p}\ (\mathsf{I}_*\ \mathsf{x}) \to \mathsf{fold}\ \varphi\ \mathsf{p}\ \mathsf{x}) \circ \mathsf{from}\ \mathsf{p} \\
&\mathsf{unfold} :: \forall \varphi\ \mathsf{r}\ \mathsf{ix}.\mathsf{Fam}\ \varphi \Rightarrow \\
&\qquad (\forall \mathsf{ix}.\varphi\ \mathsf{ix} \to \mathsf{r}\ \mathsf{ix} \to \mathsf{PF}\ \varphi\ \mathsf{r}\ \mathsf{ix}) \to (\varphi\ \mathsf{ix} \to \mathsf{r}\ \mathsf{ix} \to \mathsf{ix}) \\
&\mathsf{unfold}\ \varphi\ \mathsf{p} = \mathsf{to}\ \mathsf{p} \circ \mathsf{hmap}\ (\lambda \mathsf{p}\ \mathsf{x} \to \mathsf{I}_*\ (\mathsf{unfold}\ \varphi\ \mathsf{p}\ \mathsf{x})) \circ \varphi\ \mathsf{p} \\
&\mathsf{compos} :: \forall \varphi\ \mathsf{ix}.\mathsf{Fam}\ \varphi \Rightarrow \\
&\qquad (\forall \mathsf{ix}.\varphi\ \mathsf{ix} \to \mathsf{ix} \to \mathsf{ix}) \to (\varphi\ \mathsf{ix} \to \mathsf{ix} \to \mathsf{ix}) \\
&\mathsf{compos}\ \varphi\ \mathsf{p} = \mathsf{to}\ \mathsf{p} \circ \mathsf{hmap}\ (\lambda \mathsf{p}\ (\mathsf{I}_*\ \mathsf{x}) \to \mathsf{I}_*\ (\varphi\ \mathsf{p}\ \mathsf{x})) \circ \mathsf{from}\ \mathsf{p}
\end{aligned}
$$

Intermediate Summary

The library provides

- functor combinators K, U, I, :+:, :×:, :▷:
- type family PF and classes Fam and El,
- inductively defined generic functions such as hmap,
- derived generic functions such as fold or compos.

To add a new function

- give an inductive or derived definition.

To use on a new datatype

- define PF, Fam and El instances (can be done using Template Haskell).

Applications

Applications

- Standard examples: equality, show (the latter needs constructor info).
- Type-indexed datatypes: convenient algebras for folds, the zipper.
- Others: generic rewriting (used in exercise assistants), generic selections.

Zipper: locations and contexts

A location is a position of type ix paired with a path of contexts up to the root:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Loc} :: (*_\varphi \to *) \to (*_\varphi \to *) \to *_\varphi \to *\ \textbf{where} \\
&\quad \mathsf{Loc} :: \forall \varphi\ a\ r\ ix.(\mathsf{Fam}\ \varphi, \mathsf{Zipper}\ \varphi\ (\mathsf{PF}\ \varphi)) \Rightarrow \\
&\qquad\qquad \varphi\ ix \to r\ ix \to \mathsf{Ctxs}\ \varphi\ ix\ r\ a \to \mathsf{Loc}\ \varphi\ r\ a
\end{aligned}
$$

A path of contexts is either empty, or it adds a layer.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Ctxs} :: (*_\varphi \to *) \to *_\varphi \to (*_\varphi \to *) \to *_\varphi \to *\ \textbf{where} \\
&\quad \mathsf{Empty} :: \forall a\ r.\mathsf{Ctxs}\ \varphi\ a\ r\ a \\
&\quad \mathsf{Push} :: \forall \varphi\ ix\ a\ b\ r. \\
&\qquad\qquad \varphi\ ix \to \mathsf{Ctx}\ (\mathsf{PF}\ \varphi)\ b\ r\ ix \to \mathsf{Ctxs}\ \varphi\ ix\ r\ a \to \\
&\qquad\qquad \mathsf{Ctxs}\ \varphi\ b\ r\ a
\end{aligned}
$$

Context layers

```
data family Ctx f :: *φ → (*φ → *) → *φ → *
```

There are no recursive positions in constant types:

```
data instance Ctx (K a) b r ix
data instance Ctx U b r ix
```

In a sum, we can only move to the value that is given:

```
data instance Ctx (f :+: g) b r ix = CL (Ctx f b r ix)
                                   | CR (Ctx g b r ix)
```

In a product, we can move to the left or the right component:

```
data instance Ctx (f :×: g) b r ix = C1 (Ctx f b r ix) (g r ix)
                                   | C2 (f r ix) (Ctx g b r ix)
```

Context layers, tags and recursion

A recursive position determines the type of the hole:

```
data instance Ctx (I xi) b r ix = CId (b :=: xi)
```

Tags fix the index of the context:

```
data instance Ctx (f :▷: xi) b r ix = CTag (ix :=: xi) (Ctx f b r ix)
```

Putting it all together

The class

$$\textbf{class}\ \mathsf{Fam}\ \varphi \Rightarrow \mathsf{Zipper}\ \varphi\ \mathsf{f}$$

defines navigation functions referring to Ctx.

On top of that, derived navigation functions working on Loc are defined:

```
enter :: Zipper φ (PF φ) ⇒ φ ix → ix → Loc φ I* ix
down  :: Loc φ I* ix → Maybe (Loc φ I* ix)
up    :: Loc φ I* ix → Maybe (Loc φ I* ix)
right :: Loc φ I* ix → Maybe (Loc φ I* ix)
left  :: Loc φ I* ix → Maybe (Loc φ I* ix)
```

Plus functions to update and exit.

Conclusions

- Using a fixed-point view for Haskell generic functions is possible even if multiple datatypes are involved.
- Code is available: multirec-0.3, zipper-0.2 on Hackage.
- Future work: parameterized datatypes (mostly done), functor composition (some ideas), benchmarking, more applications.