Universiteit Utrecht

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

Generic programming with fixed points for mutually recursive datatypes

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`

September 2, 2009

Datatype-generic programming

- Write functions that depend on the structure of datatypes.
- Equality, parsing, . . .
- Traversing data structures, collecting or modifying items.
- Type-indexed data types: tries, zippers.

This talk

- Yet another (datatype-)generic programming library for Haskell.
- Gives you access to recursive positions, i.e., it is easy to write a generic fold/catamorphism.
- Allows you to define type-indexed datatypes, e.g., zippers.
- Applicable to a large class of datatypes, in particular mutually recursive datatypes.

This talk

- Yet another (datatype-)generic programming library for Haskell.
- Gives you access to recursive positions, i.e., it is easy to write a generic fold/catamorphism.
- Allows you to define type-indexed datatypes, e.g., zippers.
- Applicable to a large class of datatypes, in particular mutually recursive datatypes.

What is in a generic programming library?

- Represent datatypes generically.
- Map between user types and their representations.
- Define functions based on representations.

What is in a generic programming library?

- Represent datatypes generically.
- Map between user types and their representations.
- Define functions based on representations.

We focus on the first: generic view or universe.

PolyP (Jansson and Jeuring 1997)

The first approach to generic programming in Haskell:

- Datatypes are represented as fixed points of sums of products.

Example

```
data Expr = Const Val
          | If    Expr Expr Expr
```

Example

```
data Expr     = Const  Val
              | If     Expr Expr Expr
```

As a functor:

```
data ExprF e  = ConstF Val
              | IfF    e    e    e
type Expr'    = Fix ExprF
data Fix f    = In (f (Fix f))
```

Example

$$
\begin{array}{llll}
\textbf{data}\ \mathsf{Expr} & = & \mathsf{Const} & \mathsf{Val} \\
 & \mid & \mathsf{If} & \mathsf{Expr}\ \mathsf{Expr}\ \mathsf{Expr}
\end{array}
$$

As a functor:

$$
\begin{array}{llll}
\textbf{type}\ \mathsf{ExprF}\ e & = & & \mathsf{Val} \\
 & \mid & & e \quad e \quad e \\
\textbf{type}\ \mathsf{Expr}' & = & \mathsf{Fix}\ \mathsf{ExprF} & \\
\textbf{data}\ \mathsf{Fix}\ f & = & \mathsf{In}\ (f\ (\mathsf{Fix}\ f)) &
\end{array}
$$

Example

```
data Expr    = Const  Val
             | If     Expr Expr Expr
```

As a functor:

```
type ExprF e =        Val
             +        e    e    e
type Expr'   = Fix ExprF
data Fix f   = In (f (Fix f))
```

Example

$$
\begin{aligned}
\textbf{data}\ \mathsf{Expr} &= \mathsf{Const}\ \ \mathsf{Val} \\
&\mid\ \mathsf{If}\ \ \mathsf{Expr}\ \mathsf{Expr}\ \mathsf{Expr}
\end{aligned}
$$

As a functor:

$$
\begin{aligned}
\textbf{type}\ \mathsf{ExprF}\ e &= \mathsf{Val} \\
&+\ e \times e \times e \\
\textbf{type}\ \mathsf{Expr}' &= \mathsf{Fix}\ \mathsf{ExprF} \\
\textbf{data}\ \mathsf{Fix}\ f &= \mathsf{In}\ (f\ (\mathsf{Fix}\ f))
\end{aligned}
$$

Example

```
data Expr   = Const  Val
            | If     Expr Expr Expr
```

As a functor:

```
type ExprF  =        K Val
           :+:       I :×: I :×: I
type Expr'  = Fix ExprF
data Fix f  = In (f (Fix f))
```

Combinators

```
data I         r = I r
data K a       r = K a
data U         r = U    -- for constructors with no arguments
data (f :+: g) r = L (f r) | R (g r)
data (f :×: g) r = f r :×: g r
```

Functors are of kind $* \to *$.

```
data Fix (f :: * → *) = In (f (Fix f))
```

Writing a generic function

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor (K a) where
  fmap f (K x) = K x
instance Functor I where
  fmap f (I x)  = I (f x)
-- instances for the other functor combinators
```

Writing a generic function

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor (K a) where
  fmap f (K x) = K x
instance Functor I where
  fmap f (I x)  = I (f x)
-- instances for the other functor combinators
```

```
fold :: Functor f ⇒ (f r → r) → Fix f → r
fold alg (In f) = alg (fmap (fold alg) f)
```

Summary of workflow

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

Limitation of the PolyP approach

Only regular datatypes can be represented.

```
data Expr = Const Val
          | If    Expr Expr Expr
```

Limitation of the PolyP approach

Only regular datatypes can be represented.

```
data Expr = Const Val
          | If    Expr Expr Expr
          | Bin   Expr Op Expr
```

Limitation of the PolyP approach

Only regular datatypes can be represented.

```
data Expr = Const Val
          | If    Expr Expr Expr
          | Bin   Expr Op Expr

data Op   = Add | Mul | Infix Expr | Flip Op
```

Limitation of the PolyP approach

Only regular datatypes can be represented.

```
data Expr = Const Val
          | If    Expr Expr Expr
          | Bin   Expr Op Expr

data Op   = Add | Mul | Infix Expr | Flip Op
```

Typical ASTs are not regular, but a family of several mutually recursive datatypes.

Classic attempts

```
data Expr     = Const  Val
              | If     Expr Expr Expr
```

```
data ExprF e  = ConstF Val
              | IfF    e e e
type Expr'    = Fix ExprF
```

Classic attempts

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr} &&= \mathsf{Const}\quad \mathsf{Val}\\
& &&\mid\ \mathsf{If}\qquad\ \mathsf{Expr}\ \mathsf{Expr}\ \mathsf{Expr}\\
& &&\mid\ \mathsf{Bin}\qquad \mathsf{Expr}\ \mathsf{Op}\ \mathsf{Expr}\\
&\textbf{data}\ \mathsf{Op} &&= \mathsf{Add} \mid \mathsf{Mul} \mid \mathsf{Infix}\ \mathsf{Expr} \mid \mathsf{Flip}\ \mathsf{Op}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathsf{ExprF}\ \mathsf{e}\ \mathsf{o} &&= \mathsf{ConstF}\ \mathsf{Val}\\
& &&\mid\ \mathsf{IfF}\qquad \mathsf{e}\ \mathsf{e}\ \mathsf{e}\\
& &&\mid\ \mathsf{BinF}\quad\ \ \mathsf{e}\ \mathsf{o}\ \mathsf{e}\\
&\textbf{data}\ \mathsf{OpF}\ \mathsf{e}\ \mathsf{o} &&= \mathsf{AddF} \mid \mathsf{MulF} \mid \mathsf{InfixF}\ \mathsf{e} \mid \mathsf{FlipF}\ \mathsf{o}\\
&\textbf{type}\ \mathsf{Expr}' &&= \mathsf{Fix}_{2,0}\ \mathsf{ExprF}\ \mathsf{OpF}\\
&\textbf{type}\ \mathsf{Op}' &&= \mathsf{Fix}_{2,1}\ \mathsf{ExprF}\ \mathsf{OpF}
\end{aligned}
$$

Kinds

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

Kinds

$$\begin{aligned}
\mathsf{Fix} &:: (* \to *) \to * \\[1em]
\mathsf{Fix}_{2,0} &:: (* \to * \to *) \to (* \to * \to *) \to * \\
\mathsf{Fix}_{2,1} &:: (* \to * \to *) \to (* \to * \to *) \to *
\end{aligned}$$

Kinds

$$
\begin{aligned}
\mathsf{Fix} &:: (* \to *) \to * \\[1em]
\mathsf{Fix}_{2,0} &:: (* \to * \to *) \to (* \to * \to *) \to * \\
\mathsf{Fix}_{2,1} &:: (* \to * \to *) \to (* \to * \to *) \to * \\[1em]
\mathsf{Fix}_{3,0} &:: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to * \\
\mathsf{Fix}_{3,1} &:: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to * \\
\mathsf{Fix}_{3,2} &:: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to *
\end{aligned}
$$

Kinds

$$
\begin{aligned}
&\mathsf{Fix} \quad :: (* \to *) \to * \\
\\
&\mathsf{Fix}_{2,0} :: (* \to * \to *) \to (* \to * \to *) \to * \\
&\mathsf{Fix}_{2,1} :: (* \to * \to *) \to (* \to * \to *) \to * \\
\\
&\mathsf{Fix}_{3,0} :: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to * \\
&\mathsf{Fix}_{3,1} :: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to * \\
&\mathsf{Fix}_{3,2} :: (* \to * \to * \to *) \to (* \to * \to * \to *) \to (* \to * \to * \to *) \to * \\
\\
&\ldots
\end{aligned}
$$

Kinds (contd.)

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

Kinds (contd.)

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

If we had tuples on the kind level:

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

Kinds (contd.)

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

If we had tuples on the kind level:

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

And if we had numbers as kinds:

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

Kinds (contd.)

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

If we had tuples on the kind level:

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

And if we had numbers as kinds:

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

And this can be generalized:

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

One fixed point combinator

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

Can we express n in Haskell?

One fixed point combinator

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

Can we express n in Haskell?

Yes!

Encoding kind n

- Choose $*$ rather than n.

Encoding kind n

- Choose $*$ rather than n.
- Ensure that wherever $*$ is used instead of n, we only instantiate it with one of n different types – the types that make up our family.

Encoding kind n

- Choose $*$ rather than n.
- Ensure that wherever $*$ is used instead of n, we only instantiate it with one of n different types – the types that make up our family.
- Where necessary, provide additional evidence (in the form of a GADT) that the type is actually one of only n different possibilities.

Encoding kind n

- Choose $*$ rather than n.
- Ensure that wherever $*$ is used instead of n, we only instantiate it with one of n different types – the types that make up our family.
- Where necessary, provide additional evidence (in the form of a GADT) that the type is actually one of only n different possibilities.

$\forall \mathrm{ix} :: \mathrm{n}. \ldots$

becomes

$\forall \mathrm{ix} :: *. \mathrm{Fam}\ \mathrm{ix} \to \ldots$

Example index GADT

```
data Fam :: * → * where
  Expr :: Fam Expr
  Op   :: Fam Op
```

A value of Fam t encodes a proof that t is either Expr or Op.

Representing a family

```
data ExprF   e o =
        ConstF Val
   |    IfF     e          e          e
   |    BinF    e          o          e
data OpF     e o =
        AddF  |  MulF  |  InfixF e       |  FlipF o
```

Representing a family

```
data ExprF  (r :: * → *) (ix :: *) =
        ConstF Val
    |   IfF    (r Expr)   (r Expr)   (r Expr)
    |   BinF   (r Expr)   (r Op)     (r Expr)
data OpF    (r :: * → *) (ix :: *) =
        AddF | MulF | InfixF (r Expr) | FlipF (r Op)
```

Representing a family

```
data ExprF (r :: * → *) (ix :: *) =
        ConstF Val
     |  IfF    (r Expr)  (r Expr)  (r Expr)
     |  BinF   (r Expr)  (r Op)    (r Expr)
data OpF   (r :: * → *) (ix :: *) =
        AddF | MulF | InfixF (r Expr) | FlipF (r Op)
```

```
data FamF  (r :: * → *) (ix :: *) where
        ExprF :: ExprF r Expr → FamF r Expr
     |  OpF   :: OpF r Op     → FamF r Op
type Expr′ = Fix FamF Expr
type Op′   = Fix FamF Op
```

Representing a family

$$
\begin{array}{lllll}
\textbf{type}\ \mathsf{ExprF} & = & & & \\
 & \mathsf{K\ Val} & & & \\
:+: & \mathsf{I\ Expr} & :\times: \mathsf{I\ Expr} & :\times: \mathsf{I\ Expr} & \\
:+: & \mathsf{I\ Expr} & :\times: \mathsf{I\ Op} & :\times: \mathsf{I\ Expr} & \\
\textbf{type}\ \mathsf{OpF} & = & & & \\
 & \mathsf{U} & :+: \mathsf{U} & :+: \mathsf{I\ Expr} & :+: \mathsf{I\ Op}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{FamF}\ (r :: * \to *)\ (ix :: *)\ \textbf{where} & & \\
\quad \mathsf{ExprF} :: \mathsf{ExprF}\ r\ \mathsf{Expr} & \to & \mathsf{FamF}\ r\ \mathsf{Expr} \\
\mid\ \mathsf{OpF} :: \mathsf{OpF}\ r\ \mathsf{Op} & \to & \mathsf{FamF}\ r\ \mathsf{Op} \\
\textbf{type}\ \mathsf{Expr}' = \mathsf{Fix\ FamF\ Expr} & & \\
\textbf{type}\ \mathsf{Op}' = \mathsf{Fix\ FamF\ Op} & &
\end{array}
$$

Representing a family

$$
\begin{array}{l}
\textbf{type}\ \mathsf{ExprF} \quad = \\
\qquad\qquad\qquad \mathsf{K\ Val} \\
\quad :+: \qquad\quad \mathsf{I\ Expr} \ :\times:\ \mathsf{I\ Expr} \ :\times:\ \mathsf{I\ Expr} \\
\quad :+: \qquad\quad \mathsf{I\ Expr} \ :\times:\ \mathsf{I\ Op} \ :\times:\ \mathsf{I\ Expr} \\
\textbf{type}\ \mathsf{OpF} \quad = \\
\qquad \mathsf{U} \ :+:\ \mathsf{U} \ :+:\ \mathsf{I\ Expr} \ :+:\ \mathsf{I\ Op}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathsf{FamF} \quad = \\
\qquad\qquad\qquad \mathsf{ExprF} \ :\triangleright:\ \mathsf{Expr} \\
\quad :+: \qquad\quad \mathsf{OpF} \ :\triangleright:\ \mathsf{Op} \\
\textbf{type}\ \mathsf{Expr}' = \mathsf{Fix\ FamF\ Expr} \\
\textbf{type}\ \mathsf{Op}' \ = \mathsf{Fix\ FamF\ Op}
\end{array}
$$

Combinators for functors

Recursing on a particular index

$$\mathbf{data}\ \mathsf{I}\ (ix' :: *)\ (r :: * \to *)\ (ix :: *) = \mathsf{I}\ (r\ ix')$$

Selecting a particular index

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

Generalizing Functor

$$
\begin{array}{l}
\textbf{class}\ \mathsf{HFunctor\ fam}\ (\mathsf{f} :: (* \to *) \to * \to *)\ \textbf{where} \\
\quad \mathsf{hmap} :: \forall \mathsf{r}\ \mathsf{r}'. \\
\qquad\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \quad \mathsf{r\ ix} \to \quad \mathsf{r}'\ \mathsf{ix}) \to \\
\qquad\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \mathsf{f\ r\ ix} \to \mathsf{f\ r}'\ \mathsf{ix})
\end{array}
$$

Generalizing Functor

$$
\begin{aligned}
&\textbf{class}\ \mathsf{HFunctor\ fam}\ (\mathsf{f} :: (* \to *) \to * \to *)\ \textbf{where} \\
&\quad \mathsf{hmap} :: \forall \mathsf{r}\ \mathsf{r}'. \\
&\qquad\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \mathsf{r\ ix} \to \mathsf{r}'\ \mathsf{ix}) \to \\
&\qquad\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \mathsf{f\ r\ ix} \to \mathsf{f\ r}'\ \mathsf{ix})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fold} :: \forall \mathsf{fam\ f\ r}.\mathsf{HFunctor\ fam\ f} \Rightarrow \\
&\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \mathsf{f\ r\ ix} \to \mathsf{r\ ix}) \to \\
&\qquad (\forall \mathsf{ix}.\mathsf{fam\ ix} \to \mathsf{Fix\ f\ ix} \to \mathsf{r\ ix})
\end{aligned}
$$

Universiteit Utrecht

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

In the paper or the library

Details

- Conversion between original family and representation.
- Generic function code.

In the paper or the library

Details

- Conversion between original family and representation.
- Generic function code.

Applications

- Variants of folds.
- Classic examples: show, equality.
- Type-indexed datatypes: the zipper.
- Generic rewriting.

Try it

On Hackage

multirec – library described in the paper

zipper – generic zippers based on multirec

regular – single-datatype version of the library