Generic programming with fixed points for mutually recursive datatypes

Alexey Rodriguez[1] Stefan Holdermans[1] Andres Löh[1] Johan Jeuring[1,2]

[1]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[2]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

{alexey,stefan,andres,johanj}@cs.uu.nl

Abstract

Many datatype-generic functions need access to the recursive positions in the structure of the datatype, and therefore adopt a fixed point view on datatypes. Examples include variants of fold that traverse the data following the recursive structure, or the zipper data structure that enables navigation along the recursive positions. However, Hindley-Milner-inspired type systems with algebraic datatypes make it difficult to express fixed points for anything but regular datatypes. Many real-life examples such as abstract syntax trees are in fact systems of mutually recursive datatypes and therefore excluded. Using Haskell's GADTs and type families, we describe a technique that allows a fixed-point view for systems of mutually recursive datatypes. We demonstrate that our approach is widely applicable by giving several examples of generic functions for this view, most prominently the Zipper.

1. Introduction

One of the most ubiquitous activities in software development is *structuring data*. Indeed, many programming methods and development tools center around the creation of datatypes (or XML schemas, UML models, classes, grammars, et cetera). Once the structure of the data has been decided on, a programmer adds *functionality* to the datatypes. Here, there is always some functionality that is specific to a datatype – and part of the reason that the datatype has been designed in the first place. Other functionality is, however, *generic* and similar or even the same on many datatypes. Classic examples of such generic functionality are testing for equality, ordering, parsing, and pretty printing.

Implementing generic functionality can be tiresome and, therefore, error-prone: it involves adapting and applying the same high-level programming patterns to different datatypes, time and time again. *Datatype-generic programming* alleviates this burden by enabling programmers to write *generic functions*, i.e., functions that are defined once, but that can be used on many different datatypes.

Over the years, a vast body of work has emerged on adding support for datatype-generic programming to mainstream functional programming languages, most notably Haskell (Peyton Jones 2003). While early proposals encompassed extending the underlying language with dedicated new constructs for generic programming (Jansson and Jeuring 1997; Hinze 2000a,b), recent approaches favour the definition of generic functions in Haskell itself using Haskell's advanced type-class system (Cheney and Hinze 2002; Hinze 2004; Lämmel and Peyton Jones 2003).

The various approaches to generic programming generally differ in the expressivity of the generic functions that can be defined and the classes of datatypes that are supported. The most prominent example is that quite a number of generic functions operate on the recursive structure of datatypes, but most approaches do not provide access to the recursive positions in a datatype's definition. The approaches that do provide access to these recursive positions are limited in the sense that they only apply to a restricted set of datatypes. In particular, the full recursive structure of families of mutually recursive datatypes is beyond the reach of these approaches. Still, many real-life applications of functional programming do involve mutually recursive datatypes, arguably the most striking example being the representation of abstract syntax trees in compilers. Moreover, the generic functions that arise in such applications typically require access to the full recursive structure of these types; examples include navigation (Huet 1997; Hinze et al. 2004; McBride 2008), unification (Jansson and Jeuring 1998), rewriting (Jansson and Jeuring 2000; Van Noort et al. 2008), and pattern matching (Jeuring 1995) and, more generally, recursion schemes such as *fold* and the like (Meijer et al. 1991) and downwards accumulations (Gibbons 2000).

In this paper, we present an in Haskell embedded approach to datatype-generic programming that does enable the definition of generic functions over the full recursive structure of mutually recursive datatypes. Specifically, our contributions are the following:

- We show how to generalise the encoding of regular datatypes as fixed points of functors (reviewed in Section 2) to arbitrary families of mutually recursive types. We make use of a single higher-order fixed point operator (Section 3).
- The functors for families of mutually recursive datatypes can be constructed from a small set of combinators, thereby enabling datatype-generic programming (Section 4).
- We present several applications of generic programming in this setting, most notably the Zipper for mutually recursive types in Section 5 and generic rewriting in Section 6.

Related work is presented in Section 7, and future work and conclusions in Section 8.

A strength of our approach is that it can be readily implemented in Haskell, making use of language extensions such as type families (Schrijvers et al. 2008) and GADTs (Peyton Jones et al. 2006). The multirec and zipper libraries that are based on this paper can be obtained from HackageDB.

2. Fixed points for representing regular datatypes

Let us first review generic programming using fixed points for regular datatypes. While this is well-known, it serves not only as an introduction to the terminology we are using, but also as a template for our introduction of the more general case for families of mutually recursive types in Section 4.

A functor is a datatype of kind $* \to *$ for which we can define a *map* function. Fixed points are represented by an instance of the Fix datatype:

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

Haskell's record notation is used to introduce the selector function $out :: \mathsf{Fix}\ \mathsf{f} \to \mathsf{f}\ (\mathsf{Fix}\ \mathsf{f})$.

2.1 Defining a pattern functor directly

Using Fix, we can represent the following datatype for simple arithmetic expressions

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

by its *pattern functor*:

```
data PF_Expr r = ConstF Int | AddF r r | MulF r r
type Expr' = Fix PF_Expr
```

The types Expr and Expr′ are isomorphic, and in Haskell we can witness the isomorphisms by instantiating a class

```
class Regular a where
  deepFrom :: a → Fix (PF a)
  deepTo   :: Fix (PF a) → a
```

where

```
type family PF a :: * → *
```

The type family PF is an open type-level function mapping an index a of kind $*$ to a functor PF a of kind $* \to *$. We can instantiate it by saying

```
type instance PF Expr = PF_Expr
```

The functions *deepFrom* and *deepTo* are straight-forward to define. In practice, converting between a datatype and its fixed-point representation occurs often when programming generically, and traversing the whole value as required by *deepFrom* and *deepTo* is often more work than is actually required.

We therefore present an alternative correspondence, making use of the isomorphism

$$\mathsf{a} \cong \mathsf{Fix}\ (\mathsf{PF}\ \mathsf{a}) \cong (\mathsf{PF}\ \mathsf{a})\ (\mathsf{Fix}\ (\mathsf{PF}\ \mathsf{a})) \cong (\mathsf{PF}\ \mathsf{a})\ \mathsf{a}$$

This means that we relate a to its one-layer unfolding PF a a (a *shallow* conversion). We redefine class Regular to use the following conversion functions *from* and *to*:

```
class Regular a where
  from :: a → PF a a
  to   :: PF a a → a
```

As before, in the instance for expressions,

```
instance Regular Expr where
  from = from_Expr
  to   = to_Expr
```

the shallow conversion functions $from_{\mathsf{Expr}}$ and to_{Expr} are trivial to define.

In order to establish that $\mathsf{PF}_{\mathsf{Expr}}$ really is a functor, we make it an instance of class Functor:

```
instance Functor PF_Expr where
  fmap f (ConstF i)  = ConstF i
  fmap f (AddF e e') = AddF (f e) (f e')
  fmap f (MulF e e') = MulF (f e) (f e')
```

Given *fmap*, many recursion schemes can be defined, for example:

```
fold :: (Regular a, Functor (PF a)) ⇒
        (PF a r → r) → (a → r)
fold f = f ∘ fmap (fold f) ∘ from

unfold :: (Regular a, Functor (PF a)) ⇒
          (r → PF a r) → (r → a)
unfold f = to ∘ fmap (unfold f) ∘ f
```

Note how the conversions in the class Regular allow us to work with the original datatype Expr rather than its fixed point representation Expr′. While using shallow conversion functions affects the definitions of functions using the conversion functions such as *fold* and *unfold* slightly, it is generally at least as expressive as using a deep conversion:

```
deepFrom :: (Regular a, Functor (PF a)) ⇒ a → Fix (PF a)
deepFrom = fold In

deepTo :: (Regular a, Functor (PF a)) ⇒ Fix (PF a) → a
deepTo = unfold out
```

Another recursion scheme we can define is *compos* (Bringert and Ranta 2006). Much like *fold*, it traverses a data structure and performs operations on the children. There are different variants of *compos*, the simplest is equivalent to PolyP's *mapChildren* (Jansson and Jeuring 1998): it applies a function of type $\mathsf{a} \to \mathsf{a}$ to all children. This parameter is also responsible for performing the recursive call, because *compos* itself is not recursive:

```
compos :: (Regular a, Functor (PF a)) ⇒ (a → a) → a → a
compos f = to ∘ fmap f ∘ from
```

2.2 Building functors systematically

The approach presented above still requires us to write *fmap* by hand for every datatype. Furthermore, other applications such as navigation or rewriting require functions defined on the pattern functor that cannot directly be derived from *fmap*. Thus, having to write *fmap* and such other functions manually for each datatype is undesirable. Fortunately, it is also unnecessary.

In the following, we present a fixed set of datatypes that can be used to construct pattern functors systematically:

```
data K a       r = K a
data I         r = I r
data (f :×: g) r = f r :×: g r
data (f :+: g) r = L (f r) | R (g r)
infixr 7 :×:
infixr 6 :+:
```

The type K is used to represent occurrences of constant types, such as Int and Bool, the type I represents recursive positions. Using :×:, we can combine different fields of a constructor, and with :+:, we can combine constructors.

Using the above datatypes, we can thus represent the pattern functor of Expr as follows:

```
type PF_Expr = K Int :+: (I :×: I) :+: (I :×: I)
type Expr'   = Fix PF_Expr
```

Datatypes, such as Expr, whose recursive structure can be represented by a polynomial functor (consisting of sums, products and constants) are often called *regular datatypes*. The uniform encoding allows us to define functions that work on all regular datatypes. In particular, we can now define a generic *map* function by declaring the following instances of the class Functor:

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor I where
  fmap f (I x) = I (f x)
instance Functor (K a) where
  fmap _ (K x) = K x
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (L x) = L (fmap f x)
  fmap f (R y) = R (fmap f y)
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

With these declarations, we obtain *fmap* on $\mathsf{PF}_{\mathsf{Expr}}$ for free. Similarly, we get *fmap* on all datatypes as long as we express them as fixed points of pattern functors using K, I, :×: and :+:. Using *fmap*, we get *fold*, *unfold* and *compos* for free on all these datatypes. By providing one structural representation of a datatype (the instantiations of PF and Regular), we gain access to a multitude of powerful functions, and can easily define more.

Being able to convert between the original datatype such as Expr and the fixed point Expr′ or the one-layer unfolding PF Expr Expr now becomes much more important, because for application-specific, non-generic functions, we want to be able to use the original constructor names rather than sequences of constructor applications for the representation. This is reflected in the fact that the conversion functions $from_{\mathsf{Expr}}$ and to_{Expr}, while still being entirely straight-forward, now become more verbose. Here is $from_{\mathsf{Expr}}$ as an example:

```
from_Expr :: Expr → PF Expr Expr
from_Expr (Const i)  = L      (K i)
from_Expr (Add e e') = R (L (I e :×: I e'))
from_Expr (Mul e e') = R (R (I e :×: I e'))
```

To facilitate the conversion, some generic programming languages automatically generate mappings that relate datatypes such as Expr with their structure representation counterparts (Expr′) (Jansson and Jeuring 1997; Löh 2004; Holdermans et al. 2006). In this case – if we do not want to extend the compiler – we can use a meta-programming tool such as Template Haskell (Sheard and Peyton Jones 2002) to generate the PF and Regular instances for a datatype.

3. Fixed points for mutually recursive datatypes

In Section 2, we have shown how we can generically program with regular datatypes by expressing them as fixed points of functors. In practice, one often has to deal with large families of mutually recursive datatypes, which are not regular. As an example, consider the following extended version of our Expr datatype:

```
data Expr = Const Int
          | Add   Expr Expr
          | Mul   Expr Expr
          | EVar  Var
          | Let   Decl Expr
data Decl = Var := Expr
          | Seq   Decl Decl
type Var  = String
```

We now have two datatypes that are mutually recursive, Expr and Decl. Both make use of a third type Var. In order to deal with families such as this representation of an abstract syntax tree, we will now investigate how to generalize the representation of datatypes as fixed points of functors to such families.

3.1 Fixed points for a specific number of datatypes

Expressing a family of mutually recursive datatypes as a fixed point is folklore. However, such approaches make extensive use of the fact that on paper, you can quantify over many entities that are not available for quantification in an actual programming language.

Swierstra et al. (1999) have shown how to represent a family of two mutually recursive types as a fixed point in Haskell. The idea is to introduce a different fixed point datatype that abstracts over bifunctors of kind $* \to * \to *$ rather than functors of kind $* \to *$:

```
data Fix₂ f g = In₂ (f (Fix₂ f g) (Fix₂ g f))
```

We can easily generalize this idea further and define Fix_3, Fix_4 and so on. Depending on whether we want to count Var as a full member of our abstract syntax tree family or not, we can then use either Fix_2 or Fix_3 to represent such a family as a fixed point of functors.

The problem, however, is that we can also no longer use I, K, :+: and :×: to construct functors for arbitrary families systematically. Instead, it turns out that we require new variants of I, K, :+: and :×: for each arity. In the end, we have to rework our entire generic programming machinery for each arity of family we want to support, defeating the very purpose of generic programming. Furthermore, families of datatypes can be very large, and we cannot hope that supporting a limited amount of arities will suffice in practice.

3.2 A uniform way to represent fixed points

At first, it looks like we cannot easily abstract over arities in Haskell. However, we are going to take a somewhat different view on the different fixed point combinators that will in the end enable a uniform representation that can be expressed in Haskell.

Let us look at the kinds of Fix and Fix_2 next to each other:

```
Fix  :: (* → *)                       → *
Fix₂ :: (* → * → *) → (* → * → *) → *
```

In general, for a family of n datatypes, the fixed point combinator takes n arguments. Each argument is again parameterized over n types. We thus have:

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

If we ignore for the moment that product kinds do not exist in Haskell, we can uncurry and replace $(\mathsf{x} \to)^n *$ by $\mathsf{x}^n \to *$:

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

This, however, is still not the whole story. For a single datatype, Fix is applied to a single functor. For a family of two datatypes, we have to apply Fix_2 twice to two functors. And Fix_n has to be applied n times to n functors. Each of these applications corresponds to one of the types in the family of mutually recursive types. Since we would like to describe the whole family in terms of a single fixed point combinator, it is more accurate to view Fix_n itself as an n-tuple:

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

Now the key idea is that a tuple of n types can also be described as a function that, given the index, selects the corresponding component from the tuple. Namely, if $\mathbf{n}$ denotes a type with n inhabitants, x^n is isomorphic to $\mathbf{n} \to \mathsf{x}$:

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

Reordering the arguments reveals that we have really generalized from a fixed point for kind $*$ to a fixed point for $\mathbf{n} \to *$:

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

Apart from the fact that **n** is not available in Haskell's kind system, we now have a uniform representation of a fixed-point combinator that is suitable to express arbitrary families of datatypes. Fortunately, the remaining gap is easy to bridge, as we will show in the next section.

4. Indexed fixed points in Haskell

After having presented the idea of how to get a uniform representation of fixed points, we are now going to explain how to make use of this idea in Haskell. We develop a library for generic programming with families of mutually recursive types much in the same style as we did in Section 2 for regular datatypes. We are going to use the family of abstract syntax trees from the introduction of Section 3 as our running example.

4.1 Encoding indexed fixed points in Haskell's kind system

First, we have to find a way to encode **n** in Haskell's kind system, where **n** is supposed to be a kind that has exactly n types as inhabitants. Haskell offers just one base kind, namely $*$, so we are left with little choice. However, we can simply approximate **n** by $*$ in Haskell, as long as we promise to instantiate $*$ with only n different types.

In practice, if we have a family φ with n different types, we use the types in the family themselves as the indices to instantiate such positions of $*$. In this paper, we will write $*_\varphi$ rather than $*$ for such positions in order to make it more explicit that we are using a virtual subkind of $*$ that only consists of the members of family φ. Thus, our uniform fixed-point combinator now has kind

$$\mathsf{HFix} :: ((*_\varphi \to *) \to (*_\varphi \to *)) \to (*_\varphi \to *)$$

and can be defined in Haskell as

```
data HFix (f :: (*_φ → *) → (*_φ → *)) (ix :: *_φ) =
  HIn (f (HFix f) ix)
```

In our abstract syntax tree example, we have a family that we choose to call AST with three different types, and we are going to write $*_{\mathsf{AST}}$ for the subkind of $*$ consisting only of the types Expr, Decl, and Var.

We go even further and introduce a family-specific GADT (that we also call φ) and define it such that a value of φ ix can serve as a proof that ix is a type that belongs to φ. Whenever we quantify over a variable of kind $*_\varphi$, we will pass such a value of type φ ix to make explicit that we quantify over a limited set of types.

For the example, we introduce the GADT

```
data AST :: *_AST → * where
  Expr :: AST Expr
  Decl :: AST Decl
  Var  :: AST Var
```

such that a value of AST ix serves as a proof that ix is a member of the AST family.

One example where we make use of an explicit proof is when defining a *map* function for higher-order functors. Since the type has changed, we have to define a new class

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

The function *hmap* now has a rank-2 type. The function that is mapped is quantified over all members ix of family φ. If for every index ix in φ, this function transforms an r ix into an r′ ix, then we can transform a functor with recursive calls given by r into a functor with recursive calls given by r′.

It is perhaps instructive to note that if φ is a family consisting of only one type, there will be only one choice for φ ix, and the type of *hmap* reduces to the type of *fmap* for regular functors.

Instead of using explicit proofs of type φ ix, it is sometimes helpful to use a type class

```
class El (φ :: *_φ → *) (ix :: *_φ) where
  proof :: φ ix
```

and then use an implicit class constraint El φ ix instead of a value of type φ ix.

For the AST family, we define the following instances:

```
instance El AST Expr where proof = Expr
instance El AST Decl where proof = Decl
instance El AST Var  where proof = Var
```

4.2 Defining a pattern functor directly

Before we discuss how to represent functors of families generically, let us show how we can represent our family for abstract syntax trees as a fixed point in terms of HFix directly.

The functor for AST can be defined as follows:

```
data PF_AST :: (*_AST → *) → (*_AST → *) where
  ConstF :: Int                → PF_AST r Expr
  AddF   :: r Expr → r Expr    → PF_AST r Expr
  MulF   :: r Expr → r Expr    → PF_AST r Expr
  EVarF  :: r Var              → PF_AST r Expr
  LetF   :: r Decl → r Expr    → PF_AST r Expr
  BindF  :: r Var → r Expr     → PF_AST r Decl
  SeqF   :: r Decl → r Decl    → PF_AST r Decl
  VF     :: String             → PF_AST r Var
```

The parameter r is used to denote a recursive call. At each recursive position, we apply r to the appropriate index in order to indicate the type we recurse on. Furthermore, each constructor of the functor targets a specific member of the family.

By using HFix on the pattern functor, we obtain types that are isomorphic to the original family:

```
type Expr' = HFix PF_AST Expr
type Decl' = HFix PF_AST Decl
type Var'  = HFix PF_AST Var
```

The isomorphisms can be witnessed by conversion functions once more, and for this purpose, we declare a class Family that corresponds to Regular:

```
class Family φ where
  from :: φ ix → ix → PF φ I_* ix
  to   :: φ ix → PF φ I_* ix → ix
type family PF (φ :: *_φ → *) :: (*_φ → *) → (*_φ → *)
```

Like in the class Regular, we decide to implement a shallow conversion rather than a deep conversion. For comparison, using a deep conversion, the type of *from* would be

$$\mathit{from} :: \varphi\ \mathsf{ix} \to \mathsf{ix} \to \mathsf{HFix}\ (\mathsf{PF}\ \varphi\ \mathsf{I}_*\ \mathsf{ix})\ \mathsf{ix}$$

Note that all conversion functions take a φ ix as first argument, as proof that ix is indeed a member of φ. In the pattern functor, we have to describe the type of the recursive positions by means of a datatype of kind $*_\varphi \to *$. The one-layer unfolding uses the original datatypes of the family in the recursive positions, and we express this by choosing I_*:

```
data I_* (ix :: *_φ) = I_* {unI_* :: ix}
```

The type I_* behaves as the identity on types so that recursive occurrences inside the functor are stored "as is". Although the definition of I_* is essentially the same as that of I in Section 2, we

give it a different name to highlight that we are using it conceptually at kind $*_\varphi \to *$ rather than kind $* \to *$, even though the two kinds coincide in the Haskell code.

Here is the Family instance of AST:

```
type instance PF AST = PF_AST
instance Family AST where
  from = from_AST
  to   = to_AST
```

The functions $from_{\mathsf{AST}}$ and to_{AST} are straight-forward and not given here.

We can now go on to define a HFunctor instance and subsequently recursion schemes such as *fold* and *unfold* for $\mathsf{PF}_{\mathsf{AST}}$. However, since we strive for programming generically with families of datatypes, we want to avoid having to define HFunctor manually for our family. Instead, we will try – as we have before in Section 2 – to build our functor systematically from a fixed set of datatypes.

4.3 Building functors systematically

It turns out that we can use almost the same datatypes as before to represent functors. The datatypes K, :×:, and :+: can be lifted from being parameterized over an r of kind $*$ to being parameterized over an r of kind $*_\varphi \to *$ and an index ix of kind $*_\varphi$:

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

The type I has been used to represent a recursive call. In the current situation, recursive calls can be to a specific index in the family. Therefore, I gets an additional argument xi :: $*_\varphi$ that is used to determine the recursive call to make:

```
data I (xi :: *_φ) (r :: *_φ → *) (ix :: *_φ) = I (r xi)
```

It is perhaps surprising that xi is different from ix. But where ix projects out a certain member of the family, the type of the recursive call is independent of the type we are ultimately interested in. But in fact, we have not yet a way to make use of the parameter ix anywhere. If we look at the direct definition of $\mathsf{PF}_{\mathsf{AST}}$, we see that depending on the index we choose to project out of the functor, we get different functors. Only the first five constructors of $\mathsf{PF}_{\mathsf{AST}}$ contribute to $\mathsf{PF}_{\mathsf{AST}}$ r Expr, for example.

We introduce another building block for pattern functors in order to express such constraints on the index:

```
infix 6 :▷:
data (f :▷: (xi :: *_φ)) (r :: *_φ → *) (ix :: *_φ) where
  Tag :: f r ix → (f :▷: ix) r ix
```

By tagging a functor with an index from the family, we make explicit that the tagged part only contributes to the structure of that particular member of the family.

We now have all the building blocks we need to give a structural representation of the AST pattern functor:

```
type PF_AST = K Int                :▷: Expr :+:  -- Const
            | (I Expr :×: I Expr)  :▷: Expr :+:  -- Add
            | (I Expr :×: I Expr)  :▷: Expr :+:  -- Mul
            | I Var                :▷: Expr :+:  -- EVar
            | (I Decl :×: I Expr)  :▷: Expr :+:  -- Let
            | (I Var :×: I Expr)   :▷: Decl :+:  -- :=
            | (I Decl :×: I Decl)  :▷: Decl :+:  -- Seq
            | K String             :▷: Var       -- V
```

To match the structure of the direct definition of $\mathsf{PF}_{\mathsf{AST}}$ more closely, we have chosen to tag the representation of every constructor with the index it targets. Alternatively, we could have tagged the sum of all constructors of a type just once.

If we use the structural version of $\mathsf{PF}_{\mathsf{AST}}$ in the Family instance, we have to adapt the conversion functions. Again, these are straight-forward, but lengthy. We only show $from_{\mathsf{AST}}$:

```
from_AST :: AST ix → ix → PF_AST I_* ix
from_AST Expr (Const i)   =
  L                       (Tag (K i))
from_AST Expr (Add e e')  =
  R (L                    (Tag (ci e :×: ci e')))
from_AST Expr (Mul e e')  =
  R (R (L                 (Tag (ci e :×: ci e'))))
from_AST Expr (EVar x)    =
  R (R (R (L              (Tag (ci x)))))
from_AST Expr (Let d e)   =
  R (R (R (R (L           (Tag (ci d :×: ci e))))))
from_AST Decl (x := e)    =
  R (R (R (R (R (L        (Tag (ci x :×: ci e)))))))
from_AST Decl (Seq d d')  =
  R (R (R (R (R (R (L (Tag (ci d :×: ci d'))))))))
from_AST Var  x           =
  R (R (R (R (R (R (R (Tag (K x))))))))
ci x = I (I_* x)
```

4.4 Generic *hmap*

We still have to establish that our new functor building blocks are actually higher-order functors themselves:

```
instance El φ xi ⇒ HFunctor φ (I xi) where
  hmap f (I x) = I (f proof x)
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 y) = R (hmap f y)
instance (HFunctor φ f, HFunctor φ g) ⇒
         HFunctor φ (f :×: g) where
  hmap f (x :×: y) = hmap f x :×: hmap f y
instance HFunctor φ f ⇒ HFunctor φ (f :▷: ix) where
  hmap f (Tag x) = Tag (hmap f x)
```

Despite our generalization, the code for *hmap* looks almost completely identical to the code for *fmap*. We need an additional, but trivial case for (:▷:). A slight change occurs in the case for I, where we additionally have to require that the recursive call is actually in our family via El φ xi, to be able to pass the required *proof* to the mapped function *f*.

4.5 Generic *compos*

Using *hmap*, it is easy to define *compos*:

```
compos :: (Family φ, HFunctor φ (PF φ)) ⇒
          (∀ix.φ ix → ix → ix) → φ ix → ix → ix
compos f p = to p ∘ hmap (λp → I_* ∘ f p ∘ unI_*) ∘ from p
```

The only differences to the version in Section 2 are due to the presence of explicit proof terms of type φ ix and because the actual values in the structure are now wrapped in applications of the I_* constructor.

Bringert and Ranta (2006) describe in their paper on *compos* how to define the function on families of mutually recursive datatypes. Their solution, however, requires to modify the family of datatypes and rewrite them as a single GADT. Our version of *compos* works on families of mutually recursive datatypes without modification.

As an example use of *compos*, consider the following expression:

```
example = Let ("x" := Mul (Const 6) (Const 9))
              (Add (EVar "x") (EVar "y"))
```

The following function renames all variables in *example* – note how *renameVar'* can use the type representation to take different actions for different nodes – in this case, filter out nodes of type Var.

```
renameVar :: Expr → Expr
renameVar = renameVar' Expr
  where
    renameVar' :: AST a → a → a
    renameVar' Var x = x ++ "_"
    renameVar' p     x = compos renameVar' p x
```

The call *renameVar example* yields:

```
Let ("x_" := Mul (Const 6) (Const 9))
    (Add (EVar "x_") (EVar "y_"))
```

4.6 Generic *fold*

We can also define *fold* using *hmap*. Again, the definition is very similar to the single-datatype version:

```
type Algebra φ r = ∀ix.φ ix → PF φ r ix → r ix
fold :: (Family φ, HFunctor φ (PF φ)) ⇒
        Algebra φ r → φ ix → ix → r ix
fold f p = f p ∘ hmap (λp (I* x) → fold f p x) ∘ from p
```

Using *fold* is slightly trickier than using *compos*, because we have to construct a suitable argument of type Algebra. This algebra argument involves a function operating on the pattern functor, which is itself a generically derived datatype. We therefore have to write a function that destructs a sum of products, where the fields in the products are wrapped by occurrences of K or I. It is much more natural to define an algebra by giving one function per constructor, with the functions taking as many arguments as there are fields, preferably even in a curried style.

This problem is not caused by having families of many datatypes. The generic programming language PolyP (Jansson and Jeuring 1997) has a special ad-hoc construct that helps in defining algebras in a convenient style. We can do better: in the following, we will define a type-indexed datatype (Hinze et al. 2004) for algebras, as a type family inductively defined over the structure of functors. We can then define algebras in a convenient style, and use them in a generic fold.

The type-indexed datatype Alg is defined as follows:

```
type family Alg (f :: (*φ → *) → *φ → *)
                (r :: *φ → *) (ix :: *) :: *
type instance Alg (K a)       r ix = a → r ix
type instance Alg (I xi)      r ix = r xi → r ix
type instance Alg (f :+: g)   r ix = (Alg f r ix, Alg g r ix)
type instance Alg (K a :×: g) r ix = a → Alg g r ix
type instance Alg (I xi :×: g) r ix = r xi → Alg g r ix
type instance Alg (f :▷: xi)  r ix = Alg f r xi
```

The definition shows how we want to define our algebras: Occurrences of K and I are unwrapped. An algebra on a sum is a pair of algebras on the components. In the product case, we make use of knowledge on how datatypes are built: products are always nested to the right, and the left components are always fields, either wrapped by K or I. Hence, we can give two cases that allow us to turn algebras on a product into curried functions. The case for tags simply recurses.

We then have to show that we can transform such a more convenient algebra into the form that *fold* expects. To this end, we define the generic function *apply*:

```
class Apply (f :: (*φ → *) → *φ → *) where
  apply :: Alg f r ix → f r ix → r ix
instance Apply (K a) where
  apply f (K x) = f x
instance Apply (I xi) where
  apply f (I x) = f x
instance (Apply f, Apply g) ⇒ Apply (f :+: g) where
  apply (f, g) (L x) = apply f x
  apply (f, g) (R x) = apply g x
instance Apply g ⇒ Apply (K a :×: g) where
  apply f (K x :×: y) = apply (f x) y
instance Apply g ⇒ Apply (I xi :×: g) where
  apply f (I x :×: y) = apply (f x) y
instance Apply f ⇒ Apply (f :▷: xi) where
  apply f (Tag x) = apply f x
```

We can further facilitate the construction of algebras by defining an infix operator for pairing:

```
infixr 1 &
(&) = (,)
```

As an example, let us specify an evaluator on our abstract syntax tree types using an algebra.

Because different types in our family are mapped to different results, we need another family of datatypes for the result type of our algebra:

```
data family Value a :: *
data instance Value Expr = EV (Env → Int)
data instance Value Decl = DV (Env → Env)
data instance Value Var  = VV Var
type Env = [(Var, Int)]
```

An environment maps variables to integers. Expressions can contain variables, we therefore interpret them as functions from environments to integers. Declarations can be seen as environment transformers. Variables evaluate to their names. We can now state the algebra:

```
evalAlg :: Algebra AST Value
evalAlg = const (apply
    ( (λx                → EV (const x))                -- Const
    & (λ(EV x) (EV y) → EV (λm → x m + y m))          -- Add
    & (λ(EV x) (EV y) → EV (λm → x m * y m))          -- Mul
    & (λ(VV x)           → EV (fromJust ∘ lookup x))  -- EVar
    & (λ(DV e) (EV x) → EV (λm → x (e m)))            -- Let
    & (λ(VV x) (EV v) → DV (λm → (x, v m) : m))       -- :=
    & (λ(DV f) (DV g) → DV (g ∘ f))                   -- Seq
    & (λx                → VV x)))                    -- V
```

Testing

```
eval :: Expr → Env → Int
eval x = let (EV f) = fold evalAlg Expr x in f
```

in the expression *eval example* [("y", −12)] yields 42.

4.7 Summary

We have now introduced a library for generic programming on families of mutually recursive types. The library consists of the type family PF, the classes Family and El, and the functor constructors I, K, :+:, :×:, and :▷:. Furthermore, the library contains classes and instances for a number of generic functions, such as all the HFunctor code, the definitions of *compos*, *fold* and *unfold*.

To use the library for a specific family a user has to do the following: define a GADT such as AST, instantiate the type family PF to the pattern functor, and construct Family and El instances.

This may still seem a significant amount of work, but all of this code is entirely straight-forward and can easily be automated. In fact, we have implemented the generation of most of this boilerplate code in Template Haskell, so that only the definition of the GADT and a call to a Template Haskell function remains.

Once the library is instantiated, all generic functions that are provided by the library are available for this family without any further work.

5. The Zipper

For a tree-like datatype, the Zipper (Huet 1997) is a derived data structure that allows efficient navigation through a tree, along its recursive nodes. At every moment, the Zipper keeps track of a *location*: a point of focus paired with a context that represents the rest of the tree. The focus can be moved up, down, left, and right.

For regular datatypes, it is well-known how to define Zippers generically (Hinze et al. 2004). In the following, we first show how to define a Zipper for a system of mutually recursive datatypes using our example of abstract syntax trees (Section 5.1). Then, in Section 5.2, we give a generic algorithm in terms of the representations introduced in Section 4.3.

5.1 Zipper for mutually recursive datatypes

We first give a non-generic presentation of the Zipper for abstract syntax trees as defined in Section 3.

A location is the current focus paired with context information. In a setting with multiple types, the type of the focus ix is not known – hence, we make it existential, and carry around a representation of type AST ix:

```
data Loc_AST :: *_AST → * where
  Loc :: AST ix → ix → Ctxs_AST a ix → Loc_AST a
```

The type $\mathsf{Ctxs}_{\mathsf{AST}}$ encodes context information for the focus as a path from the focus to the root of the full tree. The path is stored in a stack of context frames:

```
data Ctxs_AST :: *_AST → *_AST → * where
  Nil  :: Ctxs_AST a a
  Cons :: Ctx_AST ix b → Ctxs_AST a ix → Ctxs_AST a b
```

A context stack of type $\mathsf{Ctxs}_{\mathsf{AST}}$ a b represents a value of type a with a b-typed *hole* in it. More specifically, a stack consists of frames of type $\mathsf{Ctx}_{\mathsf{AST}}$ ix b that represent constructor applications that yield an ix-value with a hole of type b in it. The full tree that is represented by a location can be recovered by plugging the value in focus into the topmost context frame, plugging the resulting value into the next frame, and so on. For this to work, the target type ix of each context frame must be equal to the type of the hole in the remainder of the stack – as enforced by the type of *Cons*.

5.1.1 Contexts

A single context frame $\mathsf{Ctx}_{\mathsf{AST}}$ is following the structure of the types in the AST system closely.

```
data Ctx_AST :: *_AST → *_AST → * where
  AddC1  :: Expr → Ctx_AST Expr Expr
  AddC2  :: Expr → Ctx_AST Expr Expr
  MulC1  :: Expr → Ctx_AST Expr Expr
  MulC2  :: Expr → Ctx_AST Expr Expr
  EVarC  ::        Ctx_AST Expr Var
  LetC1  :: Expr → Ctx_AST Expr Decl
  LetC2  :: Decl → Ctx_AST Expr Expr
  BindC1 :: Expr → Ctx_AST Decl Var
  BindC2 :: Var  → Ctx_AST Decl Expr
  SeqC1  :: Decl → Ctx_AST Decl Decl
  SeqC2  :: Decl → Ctx_AST Decl Decl
```

The relation between $\mathsf{Ctx}_{\mathsf{AST}}$ and AST becomes even more pronounced if we also look at the directly defined pattern functor $\mathsf{PF}_{\mathsf{AST}}$ from Section 4.2. For every constructor in $\mathsf{PF}_{\mathsf{AST}}$, we have as many constructors in $\mathsf{Ctx}_{\mathsf{AST}}$ as there are recursive positions. We can descend into a recursive position. The type of the recursive position then becomes the type of the hole, the second argument of $\mathsf{Ctx}_{\mathsf{AST}}$. The other components of the original constructor are stored in the context. As an example, consider:

```
Let  ::  Decl →  Expr →          Expr
LetF :: r Decl → r Expr → PF_AST r Expr
```

We have two recursive positions. If we descend into the first, then Decl is the type of the hole, while Expr remains – and so we get

```
LetC1 :: Expr → Ctx_AST Expr Decl
```

If, however, we descend into the second position, then Expr is the type of the hole with Decl remaining:

```
LetC2 :: Decl → Ctx_AST Expr Expr
```

5.1.2 Navigation

We now define functions that move the focus, transforming a location into a new location. These functions return their result in the Maybe monad, because navigation may fail: we cannot move down from a leaf of the tree, up from the root, or right if there are no more siblings in that direction.

Moving down analyzes the current focus. For all constructors that do not build leaves, we descend into the leftmost child by making it the new focus, and by pushing an appropriate frame onto the context stack. For leaves, we return *Nothing*.

```
down :: Loc_AST ix → Maybe (Loc_AST ix)
down (Loc Expr (Add e e') cs) =
  Just (Loc Expr e (Cons (AddC1 e') cs))
down (Loc Expr (Mul e e') cs) =
  Just (Loc Expr e (Cons (MulC1 e') cs))
down (Loc Expr (EVar x)   cs) =
  Just (Loc Var  x (Cons EVarC      cs))
down (Loc Expr (Let d e)  cs) =
  Just (Loc Decl d (Cons (LetC1 e)  cs))
down (Loc Decl (x := e)   cs) =
  Just (Loc Var  x (Cons (BindC1 e) cs))
down (Loc Decl (Seq d d') cs) =
  Just (Loc Decl d (Cons (SeqC1 d') cs))
down _                        = Nothing
```

The function *right* succeeds for nodes that actually have a right sibling. The size of the context stack remains unchanged: we just replace its top element with a new frame.

```
right :: Loc_AST ix → Maybe (Loc_AST ix)
right (Loc _ e (Cons (AddC1 e') cs)) =
  Just (Loc Expr e' (Cons (AddC2 e)  cs))
right (Loc _ e (Cons (MulC1 e') cs)) =
  Just (Loc Expr e' (Cons (MulC2 e)  cs))
right (Loc _ d (Cons (LetC1 e)  cs)) =
  Just (Loc Expr e  (Cons (LetC2 d)  cs))
right (Loc _ x (Cons (BindC1 e) cs)) =
  Just (Loc Expr e  (Cons (BindC2 x) cs))
right (Loc _ d (Cons (SeqC1 d') cs)) =
  Just (Loc Decl d' (Cons (SeqC2 d)  cs))
right _                              = Nothing
```

The function *left* is very similar to *right*. Finally, the function *up* is applicable whenever the current focus is not the root of the tree, i.e., whenever the context stack is non-empty. We then analyze the top

context frame and plug in the old focus, yielding the new focus, and retain the rest of the context. The definition is omitted for reasons of space.

5.1.3 Using the Zipper

To use the Zipper, we need functions to turn syntax trees into locations, and back again. For manipulating trees, we provide an update operation that replaces the subtree in focus.

To enter the tree, we place it into the empty context:

```
enter :: AST ix → ix → Loc_AST ix
enter p e = Loc p e Nil
```

To leave, we move up as far as possible and then return the expression in focus.

```
leave :: Loc_AST Expr → Expr
leave (Loc _ e Nil) = e
leave loc           = leave (fromJust (up loc))
```

To update the tree, we pass in a function capable of modifying the current point of focus. Because the value in focus can have different types, this function needs to be parameterized by the type representation.

```
update :: (∀ix.AST ix → ix → ix) →
          Loc_AST Expr → Loc_AST Expr
update f (Loc p x cs) = Loc p (f p x) cs
```

As an example, we modify the multiplication in

```
example = Let ("x" := Mul (Const 6) (Const 9))
              (Add (EVar "x") (EVar "y"))
```

To combine the navigation and edit operations, it is helpful to make use of flipped function composition $(\ggg) :: (a \to b) \to (b \to c) \to (a \to c)$ and monadic composition $(\geqslant\!\!\!\Rightarrow) :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$. The call

```
enter Expr ⋙ down ≥=> down ≥=> right ≥=> update solve ⋙
leave ⋙ return $ example
```

with

```
solve :: AST ix → ix → ix
solve Expr _ = Const 42
solve _    x = x
```

results in

```
Just (Let ("x" := Const 42) (Add (EVar "x") (EVar "y")))
```

5.2 A generic Zipper

We now define a Zipper generically for a system of mutually recursive datatypes. We make the same steps as in the example for abstract syntax trees before.

The type definitions for locations and context stacks stay essentially the same:

```
data Loc :: (*_φ → *) → *_φ → * where
  Loc :: (Family φ, Zipper φ (PF φ)) ⇒
         φ ix → ix → Ctxs φ a ix → Loc φ a
data Ctxs :: (*_φ → *) → *_φ → *_φ → * where
  Nil  :: Ctxs φ a a
  Cons :: φ ix → Ctx (PF φ) ix b → Ctxs φ a ix → Ctxs φ a b
```

Instead of a specific proof term AST ix, we now store a generic proof term φ ix for an arbitrary family in a location. Additionally, we need a Zipper for the system φ. This condition is expressed by Zipper (PF φ) and will be explained in more detail below.

In the stack Ctxs, we also require that the types of the elements are in φ via the field φ ix.

5.2.1 Contexts

The context type is defined generically on the pattern functor of φ. We thus reuse the type family PF defined in Section 3. We have to distinguish between different type constructors that make up the pattern functor, and therefore define Ctx as a datatype family:

```
data family Ctx f :: *_φ → *_φ → *
```

Like the context stack, a context frame is parameterized over both the type of the resulting index and the type of the hole.

The simple cases are for constant types, sums and products. There is a correspondence between the context of a datatype and its formal derivative (McBride 2001):

```
data instance Ctx (K a)     ix b = CK Void
data instance Ctx (f :+: g) ix b = CL (Ctx f ix b)
                                 | CR (Ctx g ix b)
data instance Ctx (f :×: g) ix b = C1 (Ctx f ix b) (g I_* ix)
                                 | C2 (f I_* ix) (Ctx g ix b)
```

For constants, there are no recursive positions, hence we produce an empty datatype, i.e., a datatype with no constructors:

```
data Void
```

For a sum, we are given either an f or a g, and compute the context of that. For a product, we can descend either left or right. If we descend into f, we pair a context for f with g. If we descend into g, we pair f with a context for g.

We are left with the cases for I and (:▷:). According to the analogy with the derivative, the context of the identity should be the unit type. However, we are in a situation where there are multiple types involved. The type index of I fixes the type of the hole. We express this type equality as follows, by means of a GADT:[1]

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

For the case of tags, we have a similar situation. A tag does not affect the structure of the context, it only provides information for the type system. In this case, not the type of the hole, but the type of the context itself is required to match the type index of the tag:

```
data instance Ctx (f :▷: xi) ix b where
  CTag :: Ctx f xi b → Ctx (f :▷: xi) xi b
```

This completes the definition of Ctx. We can convince ourselves that instantiating Ctx to PF AST results in a datatype that is isomorphic to $\mathsf{Ctx}_{\mathsf{AST}}$. It is also quite a bit more complex than the hand-written variant, but fortunately, the programmer never has to use it directly. Instead, we can interface with it using generic navigation functions.

5.2.2 Navigation

The navigation functions are again generically defined on the structure of the pattern functor. Thus, we define them in a class Zipper:

```
class Zipper φ f where
  ...
```

We will fill this class with methods incrementally.

Down To move down in a tree, we define a generic function *first* in our class Zipper:

[1] Currently, GHC does not allow instances of datatype families to be defined as GADTs. In the actual implementation, we therefore simulate the GADT by including an explicit proof of type equality (Peyton Jones et al. 2006; Baars and Swierstra 2002).

```
class Zipper φ f where
  ...
  first :: (∀b.φ b → b → Ctx f ix b → a) →
           f I* ix → Maybe a
```

The function takes a functor $\mathsf{f}\ \mathsf{I}_*\ \mathsf{ix}$ and tries to split off its first recursive component. This is of some type b where we know $\varphi\ \mathsf{b}$. The rest is a context of type $\mathsf{Ctx}\ \mathsf{f}\ \mathsf{ix}\ \mathsf{b}$. The function takes a continuation parameter that describes what to do with the two parts. Function *down* is defined in terms of *first*:

```
down :: Loc φ ix → Maybe (Loc φ ix)
down (Loc p x cs) =
  first (λp' z c → Loc p' z (Cons p c cs)) (from p x)
```

We try to split the tree in focus x. If this succeeds, we get a new focus z and a new context frame c. We push c on the stack.

We define *first* by induction on the structure of pattern functors. Constant types constitute the leaves in the tree. We cannot descend, and return *Nothing*.

```
instance Zipper φ (K a) where
  ...
  first f (K a) = Nothing
```

In a sum, we descend further, and add the corresponding context constructor *CL* or *CR* to the context.

```
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :+: g) where
  ...
  first f (L x) = first (λp z c → f p z (CL c)) x
  first f (R y) = first (λp z c → f p z (CR c)) y
```

We want to get to the first child. Therefore, we first try to descend to the left in a product. Only if that fails (*mplus*), we try to split the right component.

```
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :×: g) where
  ...
  first f (x :×: y) = first (λp z c → f p z (C1 c y)) x
                      `mplus` first (λp z c → f p z (C2 x c)) y
```

In the I case, we have exactly one possibility. We split $I\ (I_*\ x)$ into x and the context *CId* and pass the two parts to the continuation *f*:

```
instance El φ xi ⇒ Zipper φ (I xi) where
  ...
  first f (I (I* x)) = return (f proof x CId)
```

It is interesting to see why this is type correct: the type of x is xi, so applying f to x instantiates b to xi and forces the final argument of f to be of type $\mathsf{Ctx}\ (\mathsf{I}\ \mathsf{xi})\ \mathsf{ix}\ \mathsf{xi}$. But that is exactly the type of *CId*.

Finally, for a tag, we also descend further and apply *CTag* to the context.

```
instance Zipper φ f ⇒ Zipper φ (f :▷: xi) where
  ...
  first f (Tag x) = first (λp z c → f p z (CTag c)) x
```

This is type correct because *Tag* introduces the refinement that *CTag* requires: applying *CTag* to c results in $\mathsf{Ctx}\ (\mathsf{f} :\triangleright: \mathsf{xi})\ \mathsf{xi}\ \mathsf{b}$. This can be passed to f only if ix from the type of *first* is equal to xi. But it is, because the pattern match on *Tag* forces it to be.

Up Now that we can move down, we also want to move up again. We employ the same scheme as before: using an inductively defined generic helper function *fill*, we then define *up*. The function *fill* has the following type:

```
class Zipper φ f where
  ...
  fill :: φ b → b → Ctx f ix b → f I* ix
```

The function takes a value together with a compatible context frame and plugs them together, producing a value of the pattern functor. This operation is total, so no Maybe is required in the result.

With *fill*, we can define *up* as follows:

```
up :: Loc φ ix → Maybe (Loc φ ix)
up (Loc p x Nil)              = Nothing
up (Loc p x (Cons p' c cs)) = Just (Loc p' (to p' (fill p x c)) cs)
```

We cannot move up in the root of the tree and thus fail on an empty context stack. Otherwise, we pick the topmost context frame, and call *fill*. Since *fill* results in a value of the pattern functor, we have to convert back into the original form using *to*.

We start the definition of *fill* with the case for K. As an argument to *fill*, we need a context for K, for which we defined but one constructor *CK* with a Void parameter. In other words, in order to call *fill* on K, we have to produce a value of Void, which, apart from $\bot$, is impossible. In the context of our Zipper library, we can guarantee that $\bot$ is never produced for Void. We therefore define:

```
instance Zipper φ (K a) where
  ...
  fill p x (CK void) = impossible void
impossible :: Void → a
impossible void = error "impossible"
```

The definition of *fill* is very straight-forward: for I, we return the element to plug itself; for $(:\triangleright:)$ and $(:+:)$, we call *fill* recursively. In the case for products, we recurse into the context:

```
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :×: g) where
  ...
  fill p x (C1 c y) = fill p x c :×: y
  fill p y (C2 x c) = x :×: fill p y c
```

Right As a final example of a navigation function, we define *right*. We again employ the same scheme as before. We define a generic function *next* with the following type:

```
class Zipper φ f where
  ...
  next :: (∀b.φ b → b → Ctx f ix b → a) →
          (φ b → b → Ctx f ix b → Maybe a)
```

The function takes a context frame and an element that fits into the context. By looking at the context, it tries to move the focus one element to the right, thereby producing a new element – possibly of different type – and a new compatible context. These can, as in *first*, be combined using the passed continuation.

With *next*, we can define *right*:

```
right :: Loc φ ix → Maybe (Loc φ ix)
right (Loc p x Nil)              = Nothing
right (Loc p x (Cons p' c cs)) =
  next (λp z c' → Loc p z (Cons p' c' cs)) p x c
```

We cannot move right in the root of the tree, thus *right* fails in an empty context. Otherwise, we only need to look at the topmost context frame, and pass it to *next*, together with the current focus. On success, we take the new focus, and push the new context frame back on the stack.

Most cases of *next* are without surprises: calling *next* for K is again impossible; in sums and on tags we recurse. Since an I indicates a single child – a leaf in the tree – we cannot move right from there and return *Nothing*.

The most interesting case is the case for products. If we are currently in the first component, we try to move to the next element there, but if this fails, we have to select the first child of the second component, calling *first*. In that case, we also have to plug the old focus x back into its context c, using *fill*. If, however, we are already

in the right component, we do not need a case distinction and just try to move further to the right using *next*.

```
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :×: g) where
  ...
  next f p x   (C1 c y) =
               next (λp' z c' → f p' z (C1 c'          y )) p x c
    `mplus` first (λp' z c' → f p' z (C2 (fill p x c) c')) y
  next f p y   (C2 x c) =
               next (λp' z c' → f p' z (C2 x           c')) p y c
```

5.2.3 Using the Zipper

The functions *enter*, *leave* and *update* can be converted from the specific case for AST almost without change. The code is exactly as before, we only have to adapt the types.

```
enter :: (Family φ, Zipper φ (PF φ)) ⇒ φ ix → ix → Loc φ ix
enter p x = Loc p x Nil

leave :: Loc φ ix → ix
leave (Loc p x Nil) = x
leave loc           = leave (fromJust (up loc))

update :: (∀ix.φ ix → ix → ix) → Loc φ ix → Loc φ ix
update f (Loc p x cs) = Loc p (f p x) cs
```

Let us repeat the example from before, but now use the generic Zipper: apart from the additional argument to *enter*, nothing changes

```
enter Expr >>> down >>> down >>> right >>> update solve >>>
leave >>> return $ example
```

and the result is also the same:

```
Just (Let ("x" := Const 42) (Add (EVar "x") (EVar "y")))
```

6. Generic rewriting

Term rewriting can be specified generically, for arbitrary regular datatypes, if these are viewed as fixed points of functors (Jansson and Jeuring 2000,Van Noort et al. 2008). In the following we show how to generalize term rewriting even further, to work on families with an arbitrary number of datatypes. For reasons of space, we do not discuss generic rewriting in complete detail, but focus on the operation of matching the left-hand side of a rule with a term.

6.1 Schemes of regular datatypes

Before tackling matching on families of mutually recursive datatypes, we briefly sketch the ideas behind its implementation on regular datatypes. Consider how to implement matching for the simple version of the Expr datatype introduced in Section 2. First, we define expression schemes, which extend expressions with a constructor for rule meta-variables. Then we define matching of those schemes against expressions:

```
data ExprS = MetaVar String          | ConstS Int
           | AddS    ExprS ExprS | MulS   ExprS ExprS
match :: ExprS → Expr → Maybe [(String, Expr)]
```

On success, *match* returns a substitution mapping meta-variables to matched subterms. For example, the call

```
match (MulS (MetaVar "x") (MetaVar "y"))
      (Mul (Const 6) (Const 9))
```

yields *Just* $[(\texttt{"x"}, Const\ 6), (\texttt{"y"}, Const\ 9)]$.

To implement *match* generically, we need to define the scheme of a datatype generically. To this end, recall that a regular datatype is isomorphic to the type Fix f, for a suitably defined f. A meta-variable can appear deep inside a scheme, this suggests that the extension with *MetaVar* should take place inside the recursion, and hence on f. This motivates the following definition for schemes of regular datatypes:

```
type Scheme a = Fix (K String :+: PF a)
```

For example, the expression scheme that is used above as the first argument to *match* can be represented by

```
In (R (R (R (I (In (L (K "x")))) :×: I (In (L (K "y")))))))
```

6.2 Schemes of a datatype family and substitutions

A family of mutually recursive datatypes requires as many sorts of meta-variables as there are datatypes. For example, for the family used in Section 3, we need three meta-variables, ranging over Expr, Decl and Var, respectively. Fortunately, we can deal with all these meta-variables in one go:

```
type Scheme φ = HFix (K String :+: PF φ)
```

As in the regular case, the pattern functor is extended with a meta-variable representation. We want meta-variable representations to be polymorphic, so, unlike other constructors, K String is not tagged with (:▷:). Now, the same representation can be used to encode meta-variables that match, for example, Expr, Decl and Var.

Dealing with multiple datatypes affects the types of substitutions. We cannot use a homogeneous list of mappings as we did earlier, because different meta-variables may map to different datatypes. We get around this difficulty by existentially quantifying over the type of the matched datatype:

```
data DynIx φ = ∀ix.DynIx (φ ix) ix
type Subst φ = [(String, DynIx φ)]
```

6.3 Generic matching

Generic matching is defined as follows:

```
type MatchM s a = StateT (Subst s) Maybe a
matchM :: (Family φ, HZip φ (PF φ)) ⇒ φ ix →
          Scheme φ ix → I* ix → MatchM φ ()
matchM p (HIn (L (K metavar))) (I* e)
  = do subst ← get
       case lookup metavar subst of
         Nothing → put ((metavar, DynIx p e) : subst)
         Just _  → fail ("repeated use: " ++ metavar)
matchM p (HIn (R r)) (I* e)
  = combine matchM r (from p e)
```

Generic matching tries to match a term of type I_* ix against a scheme of corresponding type Scheme φ ix. The resulting information is returned in the MatchM monad. The definition of MatchM uses Maybe for indicating possible failure, and on top of that monad we use the state transformer StateT. The state monad is used to thread the substitution as we traverse the scheme and the term in parallel. The class HZip, which contains functionality for zipping, is introduced in the following subsection.

Generic matching consists of two cases. When dealing with a meta-variable, we first check that there is no previous mapping for it. (For the sake of brevity, we do not show how to deal with multiple occurrences of a meta-variable.) If that is the case, we update the state with the new mapping. The second case deals with matching constructors against constructors. More specifically, this corresponds to matching *Mul* (*Const* 6) (*Const* 9) against *MulS* (*MetaVar* "x") (*MetaVar* "y"). This is handled by the generic function *combine*, which matches the two pattern functor representations. If the representations match (as in our example), then *matchM* is applied to the recursive occurrences (for instance, on *MetaVar* "x" and *Const* 6, and *MetaVar* "y" and *Const* 9).

Now we can write the following wrapper on *matchM* to hide the use of the state monad that threads the substitution:

```
match :: (Family φ, HZip φ (PF φ)) ⇒ φ ix →
         Scheme φ ix → ix → Maybe (Subst φ)
match p scheme tm = execStateT (matchM p scheme (I* tm)) []
```

6.4 Generic *zip* and *combine*

The generic function *combine* is defined in terms of a another function, which is a generalization of *zipWith* for arbitrary functors. Like *hmap*, the function *hzipM* is defined by induction on the pattern functor by means of a type class:

```
class HZip φ f where
  hzipM :: Monad m ⇒
           (∀ix.φ ix → r ix → r' ix → m (r'' ix)) →
           f r ix → f r' ix → m (f r'' ix)
```

The function *hzipM* takes an argument that combines the r and r' structures stored in the pattern functor. The traversal is performed in a monad to notify failure when the functor arguments do not match, and to allow the argument to use state, for example.

In our case, we are not interested in the resulting merged structure (r'' ix). Indeed, *matchM* stores information only in the state monad, so we define *combine* to ignore the result.

```
data K* a b = K* {unK* :: a}
combine :: (Monad m, HZip φ f) ⇒
           (∀ix.φ ix → r ix → r' ix → m ()) →
           f r ix → f r' ix → m ()
combine f x y = do hzipM wrapf x y
                   return ()
  where wrapf p x y = do f p x y
                         return (K* ())
```

In the above, K_* is used to ignore the type ix in the result. The definition of *hzipM* does not differ much from that used when dealing with a single regular datatype:

```
instance El φ xi ⇒ HZip φ (I xi) where
  hzipM f (I x) (I y) = liftM I (f proof x y)
instance (HZip φ a, HZip φ b) ⇒ HZip φ (a :×: b) where
  hzipM f (x1 :×: x2) (y1 :×: y2)
    = liftM2 (:×:) (hzipM f x1 y1) (hzipM f x2 y2)
instance (HZip φ a, HZip φ b) ⇒ HZip φ (a :+: b) where
  hzipM f (L x) (L y) = liftM L (hzipM f x y)
  hzipM f (R x) (R y) = liftM R (hzipM f x y)
  hzipM f _     _     = fail "zip failed in :+:"

instance HZip φ f ⇒ HZip φ (f :▷: ix) where
  hzipM f (Tag x) (Tag y) = liftM Tag (hzipM f x y)
instance Eq a ⇒ HZip φ (K a) where
  hzipM f (K x) (K y) | x ≡ y       = return (K x)
                      | otherwise = fail "zip failed in K"
```

In the definition above, we use *liftM* and *liftM2* to turn the pure structure constructors into monadic functions.

7. Related work

Malcolm (1990) shows how to define two mutually recursive types as initial objects of functor-algebras. Swierstra et al. (1999) show how to implement fixed points for mutual recursive datatypes in Haskell. They introduce a new fixed point for every arity of mutually recursive datatypes. None of these approaches can be used as a basis for an implementation of fixed points for mutually recursive datatypes in Haskell suitable for implementing generic programs. Higher-order fixed points like our HFix have been used by Bird and Paterson (1999) and Johann and Ghani (2007) to model folds on nested datatypes.

Several authors discuss how to generate folds and other recursive schemes on mutually recursive datatypes (Böhm and Berarducci 1985; Sheard and Fegaras 1993; Swierstra et al. 1999; Lämmel et al. 2000). Again, the definitions in these papers cannot be directly generalised to families of arbitrary many datatypes in Haskell.

Mitchell and Runciman (2007) show how to obtain traversals for mutually recursive datatypes using the class Biplate. However, the type on which an action is performed remains fixed during a traversal. In contrast, the recursion schemes from Section 4.4 can apply their function arguments to subtrees of different types.

Since dependently typed programming languages have a much more powerful type system than Haskell extended with GADTs and type families, it is possible to define fixed-points for mutually recursive datatypes in many dependently typed programming languages. Benke et al. (2003) give a formal construction for mutually recursive datatypes as indexed inductive definitions in Alfa. Some similarities with our work are that the pattern functor argument is indexed by the datatype sort, and recursive positions specify the sort index of the subtree. Altenkirch and McBride (2003) show how to do generic programming in the dependently typed programming language OLEG. We believe that it is easier to write generic programs on mutually recursive datatypes in our approach, since we do not haves to deal with kind-indexed definitions, environments, type applications, datatype variables and argument variables, in addition to the cases for sums, products and constants.

McBride (2001) first described a generic Zipper on regular datatypes, which was implemented in Epigram by Morris et al. (2006). The Zipper has been used as an example of a type-indexed datatype in Generic Haskell (Hinze et al. 2004), but again only for regular datatypes. The dissection operator introduced by McBride (2008) is also only defined for regular datatypes, although McBride remarks that an implementation in a dependently typed programming language for mutually recursive datatypes is possible.

8. Conclusions

Until now, many powerful generic algorithms were known, but their adoption in practice has been hindered by their restriction to regular datatypes. In this paper, we have shown that we can overcome this restriction in a way that is directly applicable in practice: using recent extensions of Haskell, we can define generic programs that exploit the recursive structure of datatypes on families of arbitrarily many mutually recursive datatypes. For instance, extensive use of generic programming becomes finally feasible for compilers, which are often based on an abstract syntax that consists of many mutually recursive datatypes. Furthermore, our approach is non-invasive: the definitions of large families of datatypes need not be modified in order to use generic programming.

Additionally, we have demonstrated our approach by implementing several recursion schemes such as *compos* and *fold*, the Zipper, and rewriting functionality.

Based on this paper, the libraries multirec and zipper have been developed. They are available for download from HackageDB. A version of the rewriting library based on multirec will be released soon.

The multirec library contains Template Haskell code to automatically generate the boilerplate code for a family of datatypes. Also, in addition to the functionality shown here, the library offers access to the names of constructors. Furthermore, in this paper we have focused on functions that consume or transform values rather than functions that produce values. This, however, is no limitation, and we have for instance implemented a function that generates values of a particular type according to certain criteria using multirec.

In its current form, our approach cannot be used directly on parameterized types and does not support functor composition. We have, however, prototypical code that demonstrates that our approach can be extended to support these concepts without too much difficulty, and we plan to integrate this functionality into the library in the near future.

In the future, we also hope to investigate the application of our representation using $(:\triangleright:)$ to arbitrary GADTs, hopefully giving us *fold* and other generic operations on GADTs, similar to the work of Johann and Ghani (2008).

In parallel to the Haskell version, we have also experimented with an Agda (Norell 2007) version of our library, using dependent types. The Agda version has proved to be invaluable in thinking about the development without having to worry about Haskell limitations at the same time. As to Haskell, we hope that the support for type families, which we rely on very much, will continue to stabilize in the future, and that perhaps the kind system will be slightly improved with possibilities to encode kinds such as our $*_\varphi$, or with the possibility to define kind synonyms.

Acknowledgements José Pedro Magalhães and Marcos Viera commented on a previous version of this paper. Claus Reinke suggested to us the "type families desugaring trick" to get around a problem with the type checker in an older version of GHC. This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO), through its projects on "Real-life Datatype-Generic Programming" (612.063.613) and "Scriptable Compilers" (612.063.406).

References

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Kluwer, 2003.

Arthur Baars and Doaitse Swierstra. Typing dynamic typing. In *ICFP'02*, pages 157–166, 2002.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:11–2, 1999.

C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *ICFP'06*, pages 216–226, 2006.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *ACM SIGPLAN Haskell Workshop*, 2002.

Jeremy Gibbons. Generic downwards accumulations. *SCP*, 37(1–3):37–65, 2000.

Ralf Hinze. A new approach to generic functional programming. In *POPL'00*, pages 119–132. ACM Press, 2000a.

Ralf Hinze. Polytypic values possess polykinded types. In *MPC'00*, volume 1837 of *LNCS*, pages 2–27. Springer, 2000b.

Ralf Hinze. Generics for the masses. In *ICFP'04*, pages 236–243. ACM Press, 2004.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *SCP*, 51(2):117–151, 2004.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *MPC'06*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.

Gérard Huet. The Zipper. *JFP*, 7(5):549–554, 1997.

Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*, 2000.

Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482, 1997.

Patrik Jansson and Johan Jeuring. Polytypic unification. *JFP*, 8(5):527–536, 1998.

Johan Jeuring. Polytypic pattern matching. In *FPCA'95*, pages 238–248, 1995.

P. Johann and N. Ghani. Initial algebra semantics is enough! In *Proceedings, Typed Lambda Calculus and Applications*, pages 207–222, 2007.

Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *POPL'08*, pages 297–308, 2008.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, 2003.

Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In *Workshop on Generic Programming*, 2000.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

Grant Malcolm. Data structures and program transformation. *SCP*, 14: 255–279, 1990.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL'08*, pages 287–295, 2008.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. `strictlypositive.org/diff.pdf`, 2001.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *FPCA'91*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *ACM SIGPLAN Haskell Workshop*, 2007.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, LNCS. Springer, 2006.

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *ACM SIGPLAN Workshop on Generic Programming*, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP'06*, pages 50–61, 2006.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP'08*, pages 51–62. ACM Press, 2008.

Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *FPCA'93*, pages 233–242, 1993.

Tim Sheard and Simon Peyton Jones. Template meta-programming in Haskell. In *ACM SIGPLAN Haskell Workshop*, 2002.

Doaitse Swierstra, Pablo Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer, 1999.