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 important activities in software development is *structuring data*. Many programming methods and software development tools center around creating a datatype (or XML schema, UML model, class, grammar, etc.). Once the structure of the data has been designed, a software developer adds *functionality* to the datatypes. There is always some functionality that is specific for a datatype, and part of the reason why the datatype has been designed in the first place. Other functionality is similar or even the same on many datatypes. Examples of such functionality are:

- in a large datatype, looking for occurrences of a particular constructor (e.g., for representing variables) for which we want to do something, ignoring the rest of the value;
- functions that depend only on the *structure* of the datatype, such as the equality function;
- adapting the code after datatypes have changed or evolved.

Generic programming addresses these high-level programming patterns. We also use the term datatype-generic programming to distinguish the field from Java generics, Ada generic packages, generic programming in C++ STL, etc. Using generic programming, we can easily implement traversals in which a user is only interested in a small part of a possibly large value, functions which are naturally defined by induction on the structure of datatypes, and functions that automatically adapt to a changing datatype.

Generic programming grew out of category theory, in which datatypes are represented as initial algebras or fixed points of functors (Hagino 1987; Malcolm 1990; Meijer et al. 1991). A functor is a datatype of kind $* \rightarrow *$ together with a *map* function. Fixed points are represented by an instance of the Fix datatype:

```
data Fix f = In (f (Fix f))
```

and functors can be constructed using a limited set of datatypes for sums and products.

The set of datatypes that can be represented by means of Fix is limited, and does not include mutually recursive datatypes and nested datatypes (Bird and Meertens 1998). Early generic programming systems based on a fixed-point view, such as PolyP (Jansson and Jeuring 1997), inherit these restrictions. The lack of support for mutually recursive datatypes is particularly limiting, because most larger systems are described by several datatypes with complex dependencies.

Partially to overcome such limitations, several other approaches to generic programming have been developed, among them Generic Haskell (Hinze 2000; Löh 2004; Löh et al. 2008) and Scrap Your Boilerplate (Lämmel and Peyton Jones 2003). These approaches do not use fixed points to represent datatypes, and can handle mutually recursive types, and many or all nested types. However, they have no knowledge about the recursive structure of datatypes.

Quite a number of generic functions need information about the recursive structure of datatypes. Examples of such functions are the recursion schemes such as *fold* and its variants (Meijer et al. 1991), upwards and downwards accumulations (Bird et al. 1996; Gibbons 1998), unification (Jansson and Jeuring 1998), rewriting and matching functions (Jansson and Jeuring 2000), functions for selecting subexpressions (Steenbergen et al. 2008), pattern matching (Jeuring 1995), design patterns (Gibbons 2006), the Zipper and its variants (Huet 1997; McBride 2001; Hinze et al. 2004; Morris et al. 2006; McBride 2008), etc. The recursive structure of datatypes also plays an important role when transforming programs or proving properties about programs, such as the fold fusion theorem (Malcolm 1990), the generic approximation lemma (Hutton and Gibbons 2001), and the acid-rain theorem (Takano and Meijer 1995).

In a generic programming system that does not use fixed points, such functions cannot be defined properly. In Generic Haskell, views on datatypes have been added, including a fixed-point view on datatypes (Holdermans et al. 2006): a programmer can choose to write a function that does not need access to the recursive positions and works on many datatypes, or to write a function that requires the fixed-point view, but that function is again restricted to a rather small set of datatypes.

The problem of representing a system of mutually recursive datatypes by means of an initial algebra or a fixed point has been studied before (Malcolm 1990). However, such solutions cannot be used as a base for generic programming on current Hindley-Milner-inspired type systems. The number of functors in the fixed point and their arity is the same as the number of mutually recursive datatypes represented. In a language like Haskell, this pattern can only be captured by introducing new datatypes for every arity. As a consequence, we would have to duplicate the machinery over and over again, defeating the very purpose of generic programming.

This paper makes the following contributions:

- We show how to represent a system of (arbitrarily many) mutually recursive datatypes using a fixed-point view in Haskell, using extensions to the Haskell language, most prominently GADTs (Peyton Jones et al. 2006), type families (Chakravarty et al. 2005) and rank-2 types. Despite these extensions, the technique is easy to use for the programmer, and does not require more effort than other approaches to generic programming (Section 3).
- We show that our technique is widely applicable by presenting several generic algorithms that work on systems of mutually recursive datatypes. In Section 4, we demonstrate how to define recursion schemes such as *compos* (Bringert and Ranta 2006) and *fold*. In Section 5, we define a generic Zipper data structure (Huet 1997). In Section 6, we explain how to perform generic matching, which constitutes an important prerequisite for generic rewriting (Noort et al. 2008).

Our contributions are not necessarily limited to programming languages such as Haskell. We think that, in some applications, our technique may be simpler to use than those already solving the same problem in more expressive programming languages. To our knowledge, we are the first to show how to conveniently implement generic functions that need access to the recursive structure of datatypes on mutually recursive datatypes. For example, compiler writers get immediate access to generic matching, rewriting, and folding functions for their abstract syntax. The convenience is due to the lightweight nature of our approach: no generators or full dependent types are needed. Furthermore, our approach is non-invasive: the definitions of large systems of datatypes need not be modified in order to use generic programming.

In Section 7, we discuss related work, and in Section 8, we conclude. The code in this paper is executable: a Haskell module that compiles using GHC version 6.8.3 and the LaTeX file are generated from common sources.[1]

2. Fixed points for representing regular datatypes

Before we present fixed points for systems of mutually recursive datatypes, we review fixed points for regular datatypes. 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* :: Fix f → f (Fix f). 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
```

[1] Note to reviewers: until its release as a library, the code is available for download from `http://people.cs.uu.nl/andres/Rec/`

Given a *map* function for $\mathsf{PF}_{\mathsf{Expr}}$, many recursion schemes (for example, *fold* and *unfold*) can be easily defined on Expr′.

2.1 Building functors systematically

It is not necessary to define a specific *map* function for each and every pattern functor, though – as long as we build functors using a fixed set of datatypes:

```
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 :+:
```

We 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 r) = I (f r)
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
```

Now, if expressions are represented by the functor $\mathsf{PF}_{\mathsf{Expr}}$

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

we get *fmap* for expressions for free.

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. This uniform encoding allows us to define functions that work on all regular datatypes:

```
fold :: Functor f ⇒ (f a → a) → Fix f → a
fold f = f ∘ fmap (fold f) ∘ out
unfold :: Functor f ⇒ (a → f a) → a → Fix f
unfold f = In ∘ fmap (unfold f) ∘ f
```

2.2 Representations of regular datatypes

Unfortunately, functions *fold* and *unfold* can only be used on regular datatypes that are already encoded as fixed points of functors. In other words, they work on values of type Expr′, but not on values of type Expr. In practice, being forced to work with Expr′ rather than Expr is inconvenient: it is much easier to work with the user-defined, appropriately named constructors of Expr than with the structural constructors of Expr′. Therefore, 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 Haskell we can express such mappings by means of a type class. We define the type class Regular to encode the recursive structure of regular datatypes:

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

The functor type of a regular datatype is given by the *pattern functor* PF, a *type family*: for different instantiations of the type index a, we can provide different definitions of PF a. The pattern functor PF corresponds to PolyP's type constructor FunctorOf (Jansson and Jeuring 1997; Norell and Jansson 2004). The two methods *from* and *to* embed regular datatypes into their pattern functor-based representation.

Here is the Regular instance for expressions:

```
type instance PF Expr = PF_Expr
instance Regular Expr where
  from (Const i)  = L      (K i)
  from (Add e e') = R (L (I e :×: I e'))
  from (Mul e e') = R (R (I e :×: I e'))
  to (L     (K i))          = Const i
  to (R (L (I e :×: I e'))) = Add e e'
  to (R (R (I e :×: I e'))) = Mul e e'
```

Note that *from* and *to* transform only the top layer of a value. If desired, we can convert between Expr and Expr′ by recursively applying *from* and *to*.

We can now rewrite *fold* and *unfold* such that they work on instances of Regular (thus in particular, on Expr rather than Expr′):

```
fold :: Regular a ⇒ (PF a b → b) → a → b
fold f = f ∘ fmap (fold f) ∘ from
unfold :: Regular a ⇒ (b → PF a b) → b → a
unfold f = to ∘ fmap (unfold f) ∘ f
```

Another recursion scheme we can define is *compos* (Bringert and Ranta 2006). Much like *fold*, it traverses a datastructure 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 a → a to all children. This parameter is also responsible for performing the recursive call, because *compos* itself is not recursive:

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

3. Fixed points for mutually recursive datatypes

This section investigates representing systems of mutually recursive datatypes as fixed points. We first show why fixed points for single datatypes do not easily generalize to the situation of multiple datatypes. Subsequently, we present a solution to that problem, culminating in a library for representing systems of datatypes.

3.1 The problem

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 cannot represent Expr as before: the pattern functor of Expr exposes the direct recursive occurrences of Expr, but every occurrence of Decl is treated as a constant, even though declarations can contain expressions again.

One possibility, presented by Swierstra et al. (1999), is to use a different fixed point datatype, which 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))
```

Representing a set of three mutually recursive datatypes in this style requires a fixed point that takes three arguments, and so on. As a result, we cannot use one set of datatypes to build functors and define generic functions just for that one set. Instead, we basically have to duplicate the same machinery over and over again, defeating the very purpose of generic programming. Furthermore, systems of datatypes can be very large, so we cannot hope that using a limited amount of Fix-variants will suffice in practice.

3.2 Intermediate step: using a GADT

We can encode a system of (arbitrarily many) mutually recursive datatypes using a GADT (Bringert and Ranta 2006). For instance, we can replace the definitions in Section 3.1 by the GADT defined as the union of the previously separate datatypes:

```
data AST :: * → * where
  Const :: Int → AST Expr
  Add   :: AST Expr → AST Expr → AST Expr
  Mul   :: AST Expr → AST Expr → AST Expr
  EVar  :: AST Var → AST Expr
  Let   :: AST Decl → AST Expr → AST Expr
  (:=)  :: AST Var → AST Expr → AST Decl
  Seq   :: AST Decl → AST Decl → AST Decl
  V     :: String → AST Var
```

where Expr, Decl and Var are dummy types used to distinguish the categories, and, in the following, these do no longer refer to the original definitions:

```
data Expr
data Decl
data Var
```

There is a choice in the definition of AST. We can include Var in our set of mutually recursive datatypes, as we do here. Alternatively, we can just replace all occurrences of AST Var above with String, save the definition of the Var dummy type and the *V* constructor. However, we then cannot operate on variables directly in our generic functions.

It turns out that it is easier to think about the fixed point view in terms of the above GADT. In other words, we have reduced the problem of finding a fixed point for mutually recursive datatypes to that of finding a fixed point for a GADT. In the following, we "manually" define a pattern functor for the GADT, and we next show how to represent the structure of this pattern functor. Finally (in Section 3.5), we show how to use the same idea with the original system of datatypes.

3.3 Manually derived GADT pattern functor

We can define the pattern functor of our GADT directly, without using sums and products. The idea is to repeat the definition of AST given above, but now abstracting over the recursive positions.

```
data PF_AST (r :: * → *) :: * → * 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
```

A recursive occurrence is referred to by the variable r, which is indexed by the type of the subtree, and hence has kind $* \to *$.

The fixed point datatype is changed to account for the addition of the type index in the functor – but note that the kind of HFix is independent of the number of mutually recursive datatypes:

```
data HFix (f :: (∗ → ∗) → ∗ → ∗) a = HIn (f (HFix f) a)
type AST′ = HFix PF_AST
```

At the top level, the index argument a of HFix is passed to the functor f. At each recursive point, the index can be changed as desired: for example, *LetF* dictates that the index of the first recursive call is Decl, and the index of the second recursive call is Expr. Using the HFix-based encoding, the expression *Let* (*V* "x" := *Const* 2) (*EVar* (*V* "x")) can be represented by

```
HIn (LetF (HIn (BindF (HIn (VF "x")) (HIn (ConstF 2))))
          (HIn (EVarF (HIn (VF "x")))))
```

3.4 Representing the pattern functor

To support generic programming on mutually recursive datatypes in a way similar to that shown for regular ones in Section 2, we want to describe our datatypes by means of a pattern functor expressed in terms of a small number of building blocks, such as binary sums and products.

3.4.1 A failed attempt

It is not enough to just encode AST by ignoring the type index (and thus changing the kind of $\mathsf{PF}_{\mathsf{AST}}$):

```
type PF_AST = K Int     :+:   -- Const
              (I :×: I) :+:   -- Add
              (I :×: I) :+:   -- Mul
              I         :+:   -- EVar
              (I :×: I) :+:   -- Let
              (I :×: I) :+:   -- :=
              (I :×: I) :+:   -- Seq
              K String        -- V
type AST′   = Fix PF_AST
```

This way, we have a problem with the conversion between the original datatype and the pattern functor: we can write *from* (it just forgets the index), but *to* is not typeable (it has to recover the index) – consider the case for *Const*:

```
to :: AST′ → AST a
to (L (K i)) = Const i
```

The application of *Const* produces an AST Expr, but the type of *to* requires the result to be polymorphic in the index. The pattern match does not refine the type of the case, as no GADTs are involved in the match. And hence this function is not typeable. The (unsatisfactory) solution would be to make the *to* function specific to each AST category:

```
toExpr :: AST′ → AST Expr
toExpr (In (L (K i))) = Const i
```

But this would make *toExpr* partial since it cannot handle representations of AST Decl. Worse, many static guarantees, such as performing *fmap* over an expression should return an expression, would no longer be enforced by the type system.

3.4.2 A faithful representation

In order to represent the GADT AST faithfully, we have to keep track of the index, as we did in Section 3.3. We therefore perform two generalizations.

First, every type constructor gets an additional argument ix, namely, the index type of the tree being represented. Second, we add an argument xi to I to indicate the index type on which we recurse. Since the recursion may be on a different index, xi need not be the same as ix.

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

There is one missing piece in the representation. How is the choice between an Expr constructor and a Decl constructor represented? A sum can be used for the choice, but it cannot constrain ix to either type. For this purpose, we define the following GADT (:▷:). For a type expression f :▷: xi, we say that the structure representation f is tagged with the tag xi.

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

Tagging a structure representation constrains the index ix to be the same as the tag argument xi. We are now ready to give a structure representation for AST:

```
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
```

The differences with the representation of Expr in Section 2 are that I takes the index type of the recursive position, and each constructor representation is tagged with the AST category that it represents. The conversion functions hardly differ, except for dealing with tags:

```
from :: AST a → PF_AST AST a
from (Const i)  = L                    (Tag (K i))
from (Add e e’) = R (L                 (Tag (I e :×: I e’)))
from (Mul e e’) = R (R (L              (Tag (I e :×: I e’))))
from (EVar x)   = R (R (R (L           (Tag (I x)))))
from (Let d e)  = R (R (R (R (L        (Tag (I d :×: I e))))))
from (x := e)   = R (R (R (R (R (L     (Tag (I x :×: I e)))))))
from (Seq d d’) = R (R (R (R (R (R (L  (Tag (I d :×: I d’))))))))
from (V x)      = R (R (R (R (R (R (R  (Tag (K x))))))))

to :: PF_AST AST a → AST a
to (L                    (Tag (K i)))               = Const i
to (R (L                 (Tag (I e :×: I e’))))     = Add e e’
to (R (R (L              (Tag (I e :×: I e’)))))    = Mul e e’
to (R (R (R (L           (Tag (I x))))))            = EVar x
to (R (R (R (R (L        (Tag (I d :×: I e)))))))   = Let d e
to (R (R (R (R (R (L     (Tag (I x :×: I e))))))))  = x := e
to (R (R (R (R (R (R (L  (Tag (I d :×: I d’))))))))) = Seq d d’
to (R (R (R (R (R (R (R  (Tag (K x)))))))))         = V x
```

The first argument of $\mathsf{PF}_{\mathsf{AST}}$ specifies that the recursive occurrences are AST-values. Both *from* and *to* are now typeable. The pattern matches on *Tag* refine the type of the equations. For instance, in the case for *Const* the pattern match on *Tag* indicates that a must be equal to Expr on the right hand side. It follows that *Const i* is of type AST Expr.

3.5 Eliminating the GADT

Now, we turn our attention back to the AST definitions given in Section 3.1. Interestingly, we do not need the GADT AST. We use it as an inspiration to define $\mathsf{PF}_{\mathsf{AST}}$, but then map directly between $\mathsf{PF}_{\mathsf{AST}}$ and the original datatypes.

3.5.1 The library for representing systems

We can make a type class specific to our AST type to perform this mapping, but ultimately, we want a library that works with many systems of datatypes, therefore we declare:

```
class Ix s ix where
  from  :: ix → Str s ix
  to    :: Str s ix → ix
  index :: s ix
```

The parameter s indicates the system of datatypes we are working with, and the predicate Ix s ix expresses that ix is an index of system s. The structural representation of a type Str s ix is expressed in terms of a generalized pattern functor type family:

```
type family PF s :: (∗ → ∗)   -- datatype system s
                 → (∗ → ∗)   -- recursive occurrences r
                 → ∗         -- index type ix
                 → ∗
type Str s ix = (PF s) s I∗ ix
```

A datatype is represented by the pattern functor of system s constrained to represent values of type ix. Note that the r argument in Str is instantiated to the type I∗:

```
data I∗ a = I∗ {unI∗ :: a}
```

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 the different role that it plays in this representation.

The structure type constructors are extended once more, passing around the information about the system s being represented:

```
data I xi       (s :: ∗ → ∗) (r :: ∗ → ∗) ix  where
  I :: Ix s xi ⇒ r xi → I xi s r ix
data K a        (s :: ∗ → ∗) (r :: ∗ → ∗) ix = K a
data (f :+: g)  (s :: ∗ → ∗) (r :: ∗ → ∗) ix = L (f s r ix)
                                             | R (g s r ix)
data (f :×: g)  (s :: ∗ → ∗) (r :: ∗ → ∗) ix = f s r ix :×: g s r ix
data (f :▷: ix) (s :: ∗ → ∗) (r :: ∗ → ∗) ix′ where
  Tag :: f s r ix → (f :▷: ix) s r ix
```

Apart from the lifting, the only change in the type constructors is in the definition of I, where we require the type used in the recursion to be part of the datatype system, introducing the constraint Ix s xi.

It is possible to simplify this new representation by always assuming that r is I∗. As a result, we would not need the argument r and type I∗ anymore, and the argument of *I* could just have type xi rather than the current r xi. The resulting representation would be simpler to use but also more limited. While it could be used for applications such as *compos* and the Zipper, it would be useless for applications that change the type of recursive occurrences such as *fold*, *unfold*, and generic rewriting. Therefore, we prefer to use the more general representation throughout this paper.

3.5.2 Instantiating the library to a system

We now illustrate our approach by giving a representation for our datatype system and defining the conversion functions. First, we define the datatype that represents the datatypes in our system:

```
data AST :: ∗ → ∗ where
  Expr :: AST Expr
  Decl :: AST Decl
  Var  :: AST Var
```

The datatype AST fulfills two roles. First, it labels the system when used as the argument of PF. Second, the constructors of AST provide type representations on the value level, which can be used in generic functions to provide type-specific behaviour.

The pattern functor of AST is defined as follows:

```
type instance 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
```

Note that the definition of PF AST is nearly the same as the definition of $\mathsf{PF}_{\mathsf{AST}}$ before. One difference is that the kinds of the components are now different.

We can now define the conversion functions per datatype. Since the actual implementations are as before, we only show the instance for Expr as an example:

```
instance Ix AST Expr where
  from (Const i)   = L            (Tag (K i))
  from (Add e e')  = R (L         (Tag (ci e :×: ci e')))
  from (Mul e e')  = R (R (L      (Tag (ci e :×: ci e'))))
  from (EVar x)    = R (R (R (L   (Tag (ci x)))))
  from (Let d e)   = R (R (R (R (L (Tag (ci d :×: ci e))))))
  to (L            (Tag (K i)))              = Const i
  to (R (L         (Tag (e :×: e'))))        = Add (di e) (di e')
  to (R (R (L      (Tag (e :×: e')))))       = Mul (di e) (di e')
  to (R (R (R (L   (Tag x)))))               = EVar (di x)
  to (R (R (R (R (L (Tag (d :×: e))))))) = Let (di d) (di e)
  index = Expr
ci x = I (I∗ x)
di (I (I∗ x)) = x
```

The method *index* of class Ix provides access to the type representation of Expr on the value level. Such type representations will be useful later, when we define type-indexed functions.

The small GADT AST, the type family instance for PF and the instance definitions of Ix are the boilerplate code associated with our approach: the programmer has to provide this code for every system of datatypes. The boilerplate is regular enough so that it could be generated automatically, by building it into the compiler, using Template Haskell or a preprocessor, but it cannot be expressed directly within Haskell.

All the other code that we will cover now is generic for such systems of datatypes, so no further work is required in order to reap the fruits of our approach.

4. Recursion schemes

With the machinery introduced in Section 3, we can now define the recursion schemes from Section 2 for systems of mutually recursive types. Both *fold* and *compos* are based on *map*. Therefore, the key to defining generic recursion schemes is a generalization of *fmap* that we call *hmap*:

```
class HFunctor f where
  hmap :: (∀ix.Ix s ix ⇒ s ix → r ix → r′ ix) →
          f s r ix → f s r′ ix
```

This function is more general than *fmap* in two ways. First, the recursive structures being transformed (r and r′) are parametrized by an index. Second, the transforming function is polymorphic in the index type, and therefore *hmap* has a rank-2 type.

We define *hmap* by induction on the structure of the pattern functor. The only interesting instance is for I, where we apply the function parameter. In all the other instances, we just traverse the structure:

```
instance HFunctor (I xi) where
  hmap f (I x) = I (f index x)
instance HFunctor (K a) where
  hmap _ (K x) = K x
instance (HFunctor f, HFunctor g) ⇒
         HFunctor (f :+: g) where
  hmap f (L x) = L (hmap f x)
  hmap f (R y) = R (hmap f y)
instance (HFunctor f, HFunctor g) ⇒
         HFunctor (f :×: g) where
  hmap f (x :×: y) = hmap f x :×: hmap f y
instance HFunctor f ⇒ HFunctor (f :▷: ix) where
  hmap f (Tag x) = Tag (hmap f x)
```

4.1 Generic *compos*

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

```
compos :: (Ix s ix, HFunctor (PF s)) ⇒
          (∀ix.Ix s ix ⇒ s ix → ix → ix) → ix → ix
compos f = to ∘ hmap (λ ix → I∗ ∘ f ix ∘ unI∗) ∘ from
```

The only differences to the version in Section 2 are due to the presence of a type representation s 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 systems of mutually recursive datatypes. Their solution, however, requires to modify the system of datatypes and use a GADT representation such as the one in Section 3.2. Our version of *compos* works on systems of mutually recursive datatypes without modification. As an example, 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' :: Ix AST a ⇒ AST a → a → a
    renameVar' Var x = x ++ "_"
    renameVar' _   x = compos renameVar' x
```

The call *renameVar example* yields:

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

4.2 Generic *fold*

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

```
type Algebra s r = ∀ix.Ix s ix ⇒ s ix → PF s s r ix → r ix
fold :: (Ix s ix, HFunctor (PF s)) ⇒ Algebra s r → ix → r ix
fold f = f index ∘ hmap (λ _ (I∗ x) → fold f x) ∘ from
```

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 can facilitate the construction of algebras by defining suitable combinators:

```
(&) :: (a s r ix → r ix) → (b s r ix → r ix) →
       ((a :+: b) s r ix → r ix)
(f & g) (L x) = f x
(f & g) (R x) = g x
infixr 5 &
tag :: (a s r ix → r ix) → ((a :▷: ix) s r ix′ → r ix′)
tag f (Tag x) = f x
```

The (&) combinator lets us specify functions for different constructors separately, and *tag* is required to wrap tagged components. While it is possible to define more abbreviations for algebras, these two functions suffice to present an expression evaluator as an example.

Because different types in our system are mapped to different results, we need a 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:

```
evalAlgebra :: Algebra AST Value
evalAlgebra _ =
    tag (λ(K x)                   → EV (const x))
  & tag (λ(I (EV x) :×: I (EV y)) → EV (λm → x m + y m))
  & tag (λ(I (EV x) :×: I (EV y)) → EV (λm → x m ∗ y m))
  & tag (λ(I (VV x))              → EV (fromJust ∘ lookup x))
  & tag (λ(I (DV e) :×: I (EV x)) → EV (λm → x (e m)))
  & tag (λ(I (VV x) :×: I (EV v)) → DV (λm → (x, v m) : m))
  & tag (λ(I (DV f) :×: I (DV g)) → DV (g ∘ f))
  & tag (λ(K x)                   → VV x)
```

Testing

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

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

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 derived in Section 3.

5.1 Zipper for mutually recursive datatypes

We first give a non-generic presentation of the Zipper for abstract syntax trees. We use the datatypes as given in Section 3.1.

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 LocAST :: ∗ → ∗ where
  Loc :: AST ix → ix → CtxsAST a ix → LocAST a
```

The type Ctxs$_{\text{AST}}$ 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 CtxsAST :: ∗ → ∗ → ∗ where
  Empty :: CtxsAST a a
  (:.)  :: CtxAST ix b → CtxsAST a ix → CtxsAST a b
```

A context stack of type Ctxs$_{\text{AST}}$ a b represents a value of type a with a b-typed *hole* in it. More specifically, a stack consists of frames of type Ctx$_{\text{AST}}$ 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 (:.).

5.1.1 Contexts

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

```
data CtxAST :: ∗ → ∗ → ∗ where
  AddC1  :: Expr → CtxAST Expr Expr
  AddC2  :: Expr → CtxAST Expr Expr
  MulC1  :: Expr → CtxAST Expr Expr
  MulC2  :: Expr → CtxAST Expr Expr
  EVarC  ::         CtxAST Expr Var
  LetC1  :: Expr → CtxAST Expr Decl
  LetC2  :: Decl → CtxAST Expr Expr
  BindC1 :: Expr → CtxAST Decl Var
  BindC2 :: Var  → CtxAST Decl Expr
  SeqC1  :: Decl → CtxAST Decl Decl
  SeqC2  :: Decl → CtxAST Decl Decl
```

The relation between Ctx$_{\text{AST}}$ and AST becomes even more pronounced if we also look at the user-defined pattern functor PF$_{\text{AST}}$ from Section 3.3. For every constructor in PF$_{\text{AST}}$, we have as many constructors in Ctx$_{\text{AST}}$ as there are recursive positions. Into a recursive position, we can descend. The type of the recursive position then becomes the second argument of Ctx$_{\text{AST}}$. 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 → PFAST 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 → CtxAST Expr Decl
```

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

```
LetC2 :: Decl → CtxAST 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 :: LocAST ix → Maybe (LocAST ix)
down (Loc Expr (Add e e') cs) =
  Just (Loc Expr e (AddC1 e'  :. cs))
down (Loc Expr (Mul e e') cs) =
  Just (Loc Expr e (MulC1 e'  :. cs))
down (Loc Expr (EVar x)   cs) =
  Just (Loc Var  x (EVarC     :. cs))
down (Loc Expr (Let d e)  cs) =
  Just (Loc Decl d (LetC1 e   :. cs))
down (Loc Decl (x := e)   cs) =
  Just (Loc Var  x (BindC1 e  :. cs))
down (Loc Decl (Seq d d') cs) =
  Just (Loc Decl d (SeqC1 d'  :. cs))
down _                        = Nothing
```

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 first context frame and plug in the current focus.

```
up :: LocAST ix → Maybe (LocAST ix)
up (Loc _ e  (AddC1 e' :. cs)) = Just (Loc Expr (Add e e') cs)
up (Loc _ e' (AddC2 e  :. cs)) = Just (Loc Expr (Add e e') cs)
up (Loc _ e  (MulC1 e' :. cs)) = Just (Loc Expr (Mul e e') cs)
up (Loc _ e' (MulC2 e  :. cs)) = Just (Loc Expr (Mul e e') cs)
up (Loc _ x  (EVarC    :. cs)) = Just (Loc Expr (EVar x)   cs)
up (Loc _ d  (LetC1 e  :. cs)) = Just (Loc Expr (Let d e)  cs)
up (Loc _ e  (LetC2 d  :. cs)) = Just (Loc Expr (Let d e)  cs)
up (Loc _ x  (BindC1 e :. cs)) = Just (Loc Decl (x := e)   cs)
up (Loc _ e  (BindC2 x :. cs)) = Just (Loc Decl (x := e)   cs)
up (Loc _ d  (SeqC1 d' :. cs)) = Just (Loc Decl (Seq d d') cs)
up (Loc _ d' (SeqC2 d  :. cs)) = Just (Loc Decl (Seq d d') cs)
up _                           = 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 :: LocAST ix → Maybe (LocAST ix)
right (Loc _ e (AddC1 e' :. cs)) =
  Just (Loc Expr e' (AddC2 e  :. cs))
right (Loc _ e (MulC1 e' :. cs)) =
  Just (Loc Expr e' (MulC2 e  :. cs))
right (Loc _ d (LetC1 e   :. cs)) =
  Just (Loc Expr e  (LetC2 d  :. cs))
right (Loc _ x (BindC1 e :. cs)) =
  Just (Loc Expr e  (BindC2 x :. cs))
right (Loc _ d (SeqC1 d' :. cs)) =
  Just (Loc Decl d' (SeqC2 d  :. cs))
right _                          = Nothing
```

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 :: Expr → LocAST Expr
enter e = Loc Expr e Empty
```

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

```
leave :: LocAST Expr → Expr
leave (Loc _ e Empty) = 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 ix x cs) = Loc ix (f ix 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 $(\gg\!\!=\!\!>) :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$. The call

```
enter ⋙ 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 :: (∗ → ∗) → ∗ → ∗ where
  Loc :: (Ix s ix, Zipper (PF s)) ⇒ ix → Ctxs s a ix → Loc s a
data Ctxs :: (∗ → ∗) → ∗ → ∗ → ∗ where
  Empty :: Ctxs s a a
  (:.) :: Ix s ix ⇒ Ctx (PF s) s ix b → Ctxs s a ix → Ctxs s a b
```

Instead of storing a type representation in a Loc such as AST ix, we now require two things via class constraints: the type ix must be part of the system s, as expressed by Ix s ix. Furthermore, we need a Zipper for the system s. This condition is expressed by Zipper (PF s) and will be explained in more detail below.

In the stack Ctxs, we also require that the types of the elements are in s via the constraint Ix s ix.

5.2.1 Contexts

The context type is defined generically on the pattern functor of s. 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 :: (∗ → ∗)   -- datatype system s
                  → ∗          -- index type ix
                  → ∗          -- hole type b
                  → ∗
```

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)     s ix b = CK Void
data instance Ctx (f :+: g) s ix b = CL (Ctx f s ix b)
                                   | CR (Ctx g s ix b)
data instance Ctx (f :×: g) s ix b = C1 (Ctx f s ix b) (g s I∗ ix)
                                   | C2 (f s I∗ ix) (Ctx g s 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:[2]

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

```
class Zipper f where
  ...
  first :: (∀b.Ix s b ⇒ b → Ctx f s ix b → a) →
           f s I∗ ix → Maybe a
```

The function takes a functor $\mathsf{f\ s\ I_*\ ix}$ and tries to split off its first recursive component. This is of some type b with Ix s b. The rest is a context of type Ctx f s ix 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 s ix → Maybe (Loc s ix)
down (Loc x cs) = first (λz c → Loc z (c :. cs)) (from 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 (K a) where
  ...
  first f (K a) = Nothing
```

[2] 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).

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

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :+: g) where
  ...
  first f (L x) = first (λz c → f z (CL c)) x
  first f (R y) = first (λz c → f 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 f, Zipper g) ⇒ Zipper (f :×: g) where
  ...
  first f (x :×: y) = first (λz c → f z (C1 c y)) x
              `mplus` first (λz c → f 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 Zipper (I xi) where
  ...
  first f (I (I∗ x)) = return (f x CId)
```

It is interesting to see why this types: the type of *x* is xi, so applying *f* to *x* instantiates b to xi and forces the second argument of *f* to be of type Ctx (I xi) s ix 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 f ⇒ Zipper (f :▷: xi) where
  ...
  first f (Tag x) = first (λz c → f z (CTag c)) x
```

This types because *Tag* introduces the refinement that *CTag* requires: applying *CTag* to *c* results in Ctx (f :▷: xi) s xi 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 f where
  ...
  fill :: Ix s b ⇒ b → Ctx f s ix b → f s 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 s ix → Maybe (Loc s ix)
up (Loc x Empty)    = Nothing
up (Loc x (c :. cs)) = Just (Loc (to (fill 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 give the instances for *fill*, starting with 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 ⊥, is impossible. In the context of our Zipper library, we can guarantee that ⊥ is never produced for Void. We therefore define:

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

Nothing interesting happens in the sum case. We simply call *fill* recursively on the branch we are in:

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :+: g) where
  ...
  fill x (CL c) = L (fill x c)
  fill x (CR c) = R (fill x c)
```

For products we also fill recursively, on the argument indicated by the context:

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

For I, we return the element to plug itself, wrapped by the appropriate constructors:

```
instance Zipper (I xi) where
  ...
  fill x CId = I (I∗ x)
```

Again, this only types because of the refinement introduced by *CId*: the *x* is of type b, so $I\ (I_*\ x)$ would normally be of type I b s I_* ix, not I xi s I_* ix. But pattern matching on *CId* forces b and xi to be equal.

For a tag, we call *fill* recursively on the tagged context:

```
instance Zipper f ⇒ Zipper (f :▷: xi) where
  ...
  fill x (CTag c) = Tag (fill x c)
```

Once more, the refinement introduced by *CTag* is required for the use of *Tag* to be correct.

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 f where
  ...
  next :: (∀b.Ix s b ⇒ b → Ctx f s ix b → a) →
          (Ix s b ⇒ b → Ctx f s 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 s ix → Maybe (Loc s ix)
right (Loc x Empty)    = Nothing
right (Loc x (c :. cs)) = next (λz c' → Loc z (c' :. cs)) 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.

The case *next* for K is again impossible:

```
instance Zipper (K a) where
  ...
  next f x (CK void) = impossible void
```

In the case for sums we just call *next* recursively:

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :+: g) where
  ...
  next f x (CL c) = next (λz c → f z (CL c)) x c
  next f y (CR c) = next (λz c → f z (CR c)) y c
```

The case for products is the most interesting one. 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 f, Zipper g) ⇒ Zipper (f :×: g) where
  ...
  next f x (C1 c y) = next (λz c' → f z (C1 c'        y )) x c
                 `mplus` first (λz c' → f z (C2 (fill x c) c')) y
  next f y (C2 x c) = next (λz c' → f z (C2 x          c')) y c
```

Since I represents a single child, we cannot move right in such a location:

```
instance Zipper (I xi) where
  ...
  next f x CId = Nothing
```

On a tagged type, we recurse:

```
instance Zipper f ⇒ Zipper (f :▷: xi) where
  ...
  next f x (CTag c) = next (λz c' → f z (CTag c')) x 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. We have to adapt the types and respect the fact that *Loc* no longer carries a type representation. Instead, we must add a type representation as an argument to *enter* to help the type checker to associate a system of datatypes s with the type ix.

```
enter :: (Ix s ix, Zipper (PF s)) ⇒ s ix → ix → Loc s ix
enter _ x = Loc x Empty
leave :: Loc s ix → ix
leave (Loc x Empty) = x
leave loc           = leave (fromJust (up loc))
update :: (∀ix.Ix s ix ⇒ s ix → ix → ix) →
          Loc s ix → Loc s ix
update f (Loc x cs) = Loc (f index 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; Noort et al. 2008). In the following we show how to generalize term rewriting even further, to work on systems 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 systems 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* [("x", *Const* 6), ("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 system and substitutions

A system of mutually recursive datatypes requires as many sorts of meta-variables as there are datatypes. For example, for the system 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 s = HFix ((K String :+: PF s) s)
```

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 s = ∀ix.Ix s ix ⇒ DynIx (s ix) ix
type Subst s = [(String, DynIx s)]
```

6.3 Generic matching

Generic matching is defined as follows[3]:

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

[3] Currently, GHC does not unify the types PF s s ix and PF s' s' ix, even if it unifies the equivalent types that use equality constraints. This problem causes GHC 6.8.3 to reject *matchM*. To make *matchM* typeable, the actual implementation desugars the type of *from* to (Ix s ix, a∼PF s) ⇒ ix → a s I_* ix.

Generic matching tries to match a term (I_* ix) against a scheme of that type (Scheme s 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.

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 :: (HZip (PF s), Ix s ix)
      ⇒ Scheme s ix → ix → Maybe (Subst s)
match scheme tm = execStateT (matchM 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 f where
  hzipM :: Monad m ⇒
           (∀ix.Ix s ix ⇒ s ix → r ix → r′ ix → m (r″ ix)) →
           f s r ix → f s r′ ix → m (f s 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 f) ⇒
           (∀ix.Ix s ix ⇒ r ix → r′ ix → m ()) →
           f s r ix → f s r′ ix → m ()
combine f x y = do hzipM wrapf x y
                   return ()
  where wrapf _ x y = do f 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 HZip (I xi) where
  hzipM f (I x) (I y) = liftM I (f index x y)
instance (HZip a, HZip b) ⇒ HZip (a :×: b) where
  hzipM f (x1 :×: x2) (y1 :×: y2)
    = liftM2 (:×:) (hzipM f x1 y1) (hzipM f x2 y2)
instance (HZip a, HZip b) ⇒ HZip (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 f ⇒ HZip (f :▷: ix) where
  hzipM f (Tag x) (Tag y) = liftM Tag (hzipM f x y)
instance Eq a ⇒ HZip (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.

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). The definitions in these papers cannot be directly translated to Haskell because they require (type level) induction on the number of datatypes involved.

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 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 systems 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 systems of datatypes need not be modified in order to use generic programming.

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

The code for this paper is available and will be released as a Haskell library soon.

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

Acknowledgements José Pedro Magalhães and Marcos Viera commented on a previous version of this paper. 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 Lambert Meertens. Nested datatypes. In *MPC'98*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.

Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *JFP*, 6(1):1–28, 1996.

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.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP'05*, pages 241–253, 2005.

Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming*, 2006.

Jeremy Gibbons. Polytypic downwards accumulations. In *MPC'98*, volume 1422 of *LNCS*, pages 207–233. Springer-Verlag, 1998.

T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

Ralf Hinze. Polytypic values possess polykinded types. In *MPC'02*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 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-Verlag, 2006.

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

Graham Hutton and Jeremy Gibbons. The Generic Approximation Lemma. *Information Processing Letters*, 79(4):197–201, 2001.

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

Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*, pages 33–45, 2000.

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.

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, 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.

Andres Löh, Johan Jeuring, Thomas van Noort, Alexey Rodriguez, Dave Clarke, Ralf Hinze, and Jan de Wit. The Generic Haskell user's guide, Version 1.80 - Emerald release. Technical Report UU-CS-2008-011, Utrecht University, 2008.

Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 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-Verlag, 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-Verlag, 2006.

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

Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages*, volume 3145 of *LNCS*, pages 168–184. Springer-Verlag, 2004.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP'06*, pages 50–61, 2006.

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

Martijn van Steenbergen, Jeroen Leeuwestein, Johan Jeuring, José Pedro Magalhães, and Sylvia Stuurman. Selecting (sub)expressions – generic programming without generic programs. Unpublished, 2008.

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-Verlag, 1999.

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *FPCA'95*, pages 306–313, 1995.