Generic selections of subexpressions

Martijn van Steenbergen José Pedro Magalhães Johan Jeuring
Universiteit Utrecht
{msteenbe,jpm,johanj}@cs.uu.nl

Abstract

Tools for computer languages need position information: compilers for providing better error messages, structure editors for mapping between structural and textual views, and debuggers for navigating through a term, for instance. Manually adding position information to an abstract syntax tree is tedious and requires pervasive changes: the original tree becomes verbose and every function operating on it needs to be adapted.

In this paper, we describe how to automatically extend datatypes with position information using datatype-generic programming techniques. Furthermore, we show examples of how to use this position information: parsers that automatically construct trees annotated with positions, catamorphisms that deal with failure by reporting error locations, and zippers that efficiently navigate annotated trees. The generic programming technique we describe is applicable to a wide range of domains.

Keywords datatype-generic programming, Haskell, selections

1. Introduction

Most computer applications deal with some form of structured data. When this data needs to be manipulated by a user, it is often presented unstructured (in textual form), as this provides a convenient and flexible way of editing data. Internally, the application has to maintain a relation between the structural and the textual view. Typically, syntax trees are annotated with position information at every node, relating it to a location in the textual view. For example, consider an abstract syntax tree (AST) to represent simple arithmetic expressions over integers:

```
data Expr_Bare = Num Int
               | Add Expr_Bare Expr_Bare
               | Sub Expr_Bare Expr_Bare
               | Mul Expr_Bare Expr_Bare
               | Div Expr_Bare Expr_Bare
```

To add position information to our expressions we could add an additional argument to each constructor indicating the position. Alternatively, we can change every recursive occurrence to contain this information:

```
type Expr_Pos  = (Bounds, Expr_Pos')
data Expr_Pos' = Num Int
               | Add Expr_Pos Expr_Pos
               | Sub Expr_Pos Expr_Pos
               | Mul Expr_Pos Expr_Pos
               | Div Expr_Pos Expr_Pos
```

This change, albeit mechanical, is not trivial, since we cannot reuse $\mathsf{Expr}_{\mathsf{Bare}}$ in our definition of $\mathsf{Expr}_{\mathsf{Pos}}$. This means functions operating on expressions will need to be adapted. Parsers, for example, have to request the location information from the parse state and inject it into the model whenever a new node is constructed, distracting from the actual parsing process and requiring the distinction between annotated and unannotated expressions. Consumers of the new expressions need to be adapted as well, either to explicitly ignore the position if it is not needed or to use it in the results.

Implementing these changes is not difficult, but it is definitely a lot of work. It is somewhat like a design pattern: a solution to a common problem that is usually not expressed as a code library or framework, but rather as a series of descriptive steps. Our goal is to turn this design pattern into a library, and minimize the overall effort required from the programmer, in and for the functional programming language Haskell (Peyton Jones et al. 2003).

As an example of using our library, here is an application of a consumer of expressions:

```
> compileExpr exprEval "1 + 2/0 + 3 + 4/0"
Errors:
*  4- 7: division by zero
* 14-17: division by zero
```

Function *compileExpr* parses an expression, and evaluates the resulting parse tree using the algebra *exprEval* defined below. It fails on division by zero, automatically prefixing errors with the location of their occurrence in the source text.

```
exprEval expr = case expr of
  Num n   → Right n
  Add x y → Right (x + y)
  Sub x y → Right (x − y)
  Mul x y → Right (x ∗ y)
  Div x y | y ≡ 0     → Left "division by zero"
          | otherwise → Right (x ‘div‘ y)
```

This example, together with all the library code, is available in the `Annotations` package on Hackage[1].

This paper makes the following contributions:

- Using fixpoint views, we define a type-indexed datatype to recursively insert position information in algebraic datatypes (Section 3).
- We show how consumers of datatypes expressed as catamorphisms can be made to work automatically with both the bare datatypes and the derived annotated datatypes (Section 4). Fur-

[Copyright notice will appear here once 'preprint' option is removed.]

[1] `http://hackage.haskell.org/package/Annotations-0.1`

thermore, we introduce a new kind of catamorphism that makes the possibility of failure explicit, allowing automatic extraction of the relevant position information in case of failure (Section 5).

- We redefine parser combinators so that they automatically build recursively annotated trees (Section 6).
- We solve several common editor use cases, such as mapping from text selection to structural selection and back, and fixing invalid selections (Section 7).
- The solutions are presented again using the `multirec` generic programming library and compared to the previous solutions (Section 8).

After presenting our library, we discuss related work in Section 9, point directions for future research in Section 10, and conclude.

2. Representing textual selections

Text selections play an important role throughout this paper. In order to represent text selections, we first need to represent a single position within a text. There are two representations of text position that are used often: offset from the start of the text and line-column numbers. Both are useful in different circumstances. Code editors usually present the programmer with line and column numbers, because code tends to be line-oriented. However, behind the scenes programs usually work with offsets, especially if they need to do computations on these offsets. Only when presenting the position information to the user are the offsets converted to line-column numbers. Offsets are easier to work with because they do not depend on the exact characters in the input and can be expressed as a single number instead of two.

We choose offsets to represent text positions, as these are easier to work with. A text selection is then a tuple of two offsets. We call the type of text selections a Range:

```
type Range = (Int, Int)
```

Offsets can be thought of as positions between two characters (starting at zero). For any text of length n, there are $n+1$ valid offsets, namely those in the closed interval $[0..n]$. For ranges $(left, right)$ we always maintain the invariant that $0 \leqslant left$ and $left \leqslant right$. We define some utility functions to work with offsets and ranges:

```
posInRange :: Int → Range → Bool
posInRange pos (left, right) = left ⩽ pos ∧ pos ⩽ right

rangeInRange :: Range → Range → Bool
rangeInRange (left, right) range = left  `posInRange` range
                                 ∧ right `posInRange` range
```

Function *posInRange* tells whether a range contains a certain position. Ranges are closed intervals: positions may coincide with a range's end points. Function *rangeInRange inner outer* tells whether *inner* is a subrange of *outer*. Again, end points may coincide.

To be able to map between text selections and tree selections, we need to remember what each subtree's position in the input was. To translate from tree selection to text selection it is sufficient to store a Range with every node. However, if we store a single range for each node, when translating a text selection back to a tree selection the user needs to select the exact range for the selection to be recognized. It is often the case that a structure is surrounded by some whitespace in the source text; it seems only fair to allow the user to select some of this whitespace in addition to the structure itself. For that reason, we store not two but four offsets for each subtree: the point where the whitespace before the structure starts, the point where the whitespace ends and the text starts, the point where the whitespace after the structure starts, and the point where this whitespace ends. Figure 1 shows these four offsets in the node representing the literal 45 in a sample expression. Using this information, we can be more flexible and accept any text selection that starts between the first two offsets and ends between the last two offsets.

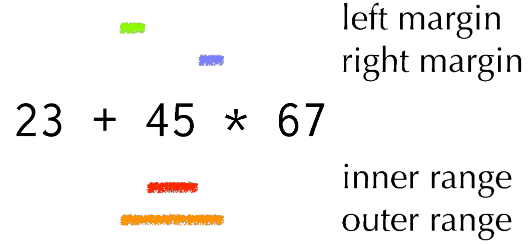

Figure 1. Illustration of inner and outer ranges.

We store the combination of these four offsets in the datatype Bounds:

```
data Bounds = Bounds { leftMargin  :: Range
                     , rightMargin :: Range }

innerRange :: Bounds → Range
innerRange (Bounds (_, left) (right, _)) = (left, right)

outerRange :: Bounds → Range
outerRange (Bounds (left, _) (_, right)) = (left, right)
```

The bounds store the left and right margin. We could store the inner and outer range just as well, in which case *leftMargin* and *rightMargin* would have been the projection functions rather than *innerRange* and *outerRange*. However, since the same margin is often shared between multiple nodes in the tree, it is more memory-efficient to store the margins than the inner and outer ranges. For example: the subtree for $45 * 67$ shares its left margin with the node for 45 and its right margin with the node for 67.

We define a function *rangeInBounds* that checks whether a range is a valid selection for a node that has the specified bounds. The left endpoint should be in the left margin, and the right endpoint should be in the right margin:

```
rangeInBounds :: Range → Bounds → Bool
rangeInBounds (l, r) b = l `posInRange` leftMargin  b
                       ∧ r `posInRange` rightMargin b
```

When using and constructing annotated parse trees we rely on some laws for ranges and bounds that we maintain as invariants. These laws are:

1. A node's inner range is always enclosed by that node's outer range, i.e. for every node's *bounds* we have that

   innerRange bounds `` `rangeInRange` `` *outerRange bounds*

2. Children appear in the same order in the source text as in the syntax tree, and their inner ranges do not overlap. For each pair of adjacent siblings, we have that their respective bounds $bounds_1$ and $bounds_2$ adhere to $fst\ (rightMargin\ bounds_1) \leqslant snd\ (leftMargin\ bounds_2)$.

3. A node's bounds always enclose that node's children's bounds. In other words, for every child c of a parent p we have that

   $innerRange\ bounds_c$ `` `rangeInRange` `` $innerRange\ bounds_p$

3. Annotated base functors

In Section 1 we have defined $\mathsf{Expr_{Pos}}$ to add position information to our expressions. This approach is unsatisfactory because we do not reuse the original datatype. In fact, $\mathsf{Expr_{Bare}}$ and $\mathsf{Expr_{Pos}}$, share the same structure: only the types of their recursive positions are different. We can increase reuse by abstracting over the parts that differ. In this case, we add a type argument to the datatype, to be filled-in later when we know whether we want a bounded or unbounded expression:

```
data Expr_F r = Num Int
              | Add r r
              | Sub r r
              | Mul r r
              | Div r r
```

A datatype which abstracts over its recursive positions in this fashion is also said to be in *open recursion* form. The $\mathsf{Expr_F}$ version of the expression datatype is often called the *base functor*.

We can now redefine $\mathsf{Expr_{Bare}}$ in terms of $\mathsf{Expr_F}$. The type argument of $\mathsf{Expr_F}$ determines the shape of its children. To reconstruct the original expression type, we want the children to be expressions as well. Therefore we need to give $\mathsf{Expr_F}$ itself as type argument to $\mathsf{Expr_F}$. This leads to the following recursive definition of $\mathsf{Expr_{Bare}}$:

```
newtype Expr_Bare = Expr_Bare (Expr_F Expr_Bare)
```

The repeated expansion of $\mathsf{Expr_{Bare}}$ leads to the infinite type $\mathsf{Expr_F}\ (\mathsf{Expr_F}\ (\mathsf{Expr_F} \ldots))$, indicating that the obtained tree is of $\mathsf{Expr_F}$-shape at every level.

$\mathsf{Expr_{Pos}}$ can be expressed in terms of $\mathsf{Expr_F}$ in a similar way. To insert the position information at every level, we wish to obtain the infinite type $(\mathsf{Bounds}, \mathsf{Expr_F}\ (\mathsf{Bounds}, \mathsf{Expr_F} \ldots))$. Again, we obtain this with a recursive datatype definition:

```
newtype Expr_Pos = Expr_Pos (Bounds, (Expr_F Expr_Pos))
```

The fact that types such as $\mathsf{Expr_{Bare}}$ and $\mathsf{Expr_{Pos}}$ use themselves as arguments to functors in their own definitions is made explicit by expressing them in terms of the well-known datatype Fix:

```
newtype Fix f     = In        { out     :: f (Fix f)  }
newtype Expr_Bare = Expr_Bare { runExpr :: Fix Expr_F }
```

Because number literals and arithmetic operators are overloaded in Haskell, we can easily construct values of the new $\mathsf{Expr_{Bare}}$ type if we supply an appropriate **instance** Num $\mathsf{Expr_{Bare}}$ and **instance** Fractional $\mathsf{Expr_{Bare}}$:

```
> runExpr (2 + 3 * 4)
In {out = Add
  (In {out = Num 2})
  (In {out = Mul (In {out = Num 3})
                 (In {out = Num 4})})}
```

To define $\mathsf{Expr_{Pos}}$ in terms of Fix we introduce a new datatype for adding position information at every level, somewhat like a tuple type that is lifted on its second argument:

```
data Ann x f a = Ann x (f a)

instance Functor f => Functor (Ann x f) where
  fmap f (Ann x t) = Ann x (fmap f t)

newtype Expr_Pos = Expr_Pos (Fix (Ann Bounds Expr_F))
```

We will use these last definitions of $\mathsf{Expr_{Bare}}$ and $\mathsf{Expr_{Pos}}$ throughout the rest of the paper. Furthermore, we introduce two type synonyms that will prove useful:

```
type AnnFix  x f = Fix (Ann x f)
type AnnFix_1 x f = f (AnnFix x f)
```

The first is a recursive tree of shape f at every level, fully annotated with x's; the second has fully annotated children but still lacks an annotation at the top level. It can be made fully annotated by providing the top-level annotation:

```
mkAnnFix :: x -> AnnFix_1 x f -> AnnFix x f
mkAnnFix x = In . Ann x
```

We have increased reuse by providing a few components we can stack and compose as necessary. This allows for generic functions; a nice example is the following function that generically removes annotations from trees. The only thing we require of the functors passed to AnnFix is that they implement the Functor type class:

```
unannotate :: Functor f => AnnFix x f -> Fix f
unannotate (In (Ann _ tree)) = In (fmap unannotate tree)
```

We omit the **instance** Functor $\mathsf{Expr_F}$ because it is trivial: there exists only a single implementation that adheres to the functor laws.

4. Catamorphisms over fixpoints

Now that we have added fields for storing position information to our expression datatype, we adapt the producers and consumers to the new expression type. We start with consumers, focusing on the particular case of catamorphisms. Before analyzing catamorphisms in the presence of position information, we review the basic concept.

To understand the concept of a catamorphism, it is instructive to look at the widely known *foldr* over lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Function *foldr* replaces the two list constructors, (:) and [], recursively by programmer-supplied functions (the first two arguments to *foldr*). These two functions together are referred to as the *algebra* for the list catamorphism. For example, *foldr* $(\oplus)$ e turns the list $x : y : z : [\,]$ into $x \oplus (y \oplus (z \oplus e))$.

Similar functions are defined for any other algebraic datatype. For the $\mathsf{Expr_{Bare}}$ datatype, for example, we define a function *cataExpr* that takes five arguments, one for each constructor. The function recursively traverses an input expression, applying the appropriate functions to the fields of the constructors. Instead of passing the five arguments separately to the function, we group them together in a special datatype capturing expression algebras:

```
data ExprAlg a = ExprAlg { cataNum :: Int    -> a
                         , cataAdd :: a -> a -> a
                         , cataSub :: a -> a -> a
                         , cataMul :: a -> a -> a
                         , cataDiv :: a -> a -> a }
```

The catamorphism for expressions becomes:

```
cataExpr :: ExprAlg a -> Expr_Bare -> a
cataExpr alg = f where
  f (In expr) = case expr of
                  Num n   -> cataNum alg n
                  Add x y -> cataAdd alg (f x) (f y)
                  Sub x y -> cataSub alg (f x) (f y)
                  Mul x y -> cataMul alg (f x) (f y)
                  Div x y -> cataDiv alg (f x) (f y)
```

Again, ExprAlg's definition is reminiscent of $\mathsf{Expr_{Bare}}$'s definition: every constructor of $\mathsf{Expr_{Bare}}$ has a corresponding constructor in ExprAlg, and each constructor in ExprAlg has fields that directly correspond to those of $\mathsf{Expr_{Bare}}$'s constructor. It turns out we have yet another use for $\mathsf{Expr_F}$: the types ExprAlg a and $\mathsf{Expr_F}\ \mathsf{a} \to \mathsf{a}$ are isomorphic. We can show this using basic algebra rules if we view the types as polynomials. Datatypes become sums (one term

per constructor) of products (one factor per constructor field) and functions $a \to b$ become exponentials b^a:

$$ExprF(a) = \mathsf{Int} + a^2 + a^2 + a^2 + a^2$$

$$\begin{aligned} ExprAlg(a) &= a^{\mathsf{Int}} * (a^a)^a * (a^a)^a * (a^a)^a * (a^a)^a \\ &= a^{\mathsf{Int}} * a^{a^2} * a^{a^2} * a^{a^2} * a^{a^2} \\ &= a^{\mathsf{Int}+a^2+a^2+a^2+a^2} \\ &= a^{\mathsf{Expr_F}(a)} \end{aligned}$$

We see that ExprAlg a and $\mathsf{Expr_F}$ a $\to$ a are isomorphic. We can now use this new type in the definition of *cataExpr*:

```
cataExpr :: (Expr_F a → a) → Expr_Bare → a
cataExpr f (In expr) = f (fmap (cataExpr f) expr)
```

This definition is significantly shorter than the previous one. This is mostly because we no longer need to pattern match on $\mathsf{Expr_F}$'s constructors, as this is hidden in the function argument and the call to *fmap*. In fact, there is nothing in *cataExpr* that is specific to $\mathsf{Expr_F}$ anymore. Therefore, its signature above is too specific. Writing the body in pointfree style, the new function becomes:

```
type Algebra f a = f a → a

cata :: Functor f ⇒ Algebra f a → Fix f → a
cata f = f ∘ fmap (cata f) ∘ out
```

The open recursion style for datatypes allows for datatype-generic programming: having abstracted from the recursive positions, we can express recursion schemes like *cata* once and for all.

As an added bonus, writing algebras in this form results in very elegant code. For example, evaluating expressions to integers (without taking division by zero into account) is implemented as follows:

```
exprEval :: Algebra Expr_F Int
exprEval expr = case expr of
                  Num n   → n
                  Add x y → x + y
                  Sub x y → x − y
                  Mul x y → x ∗ y
                  Div x y → x `div` y
```

Because *cata* takes care of the recursive positions, the algebra may assume that the fields already contain the results of the evaluation. We test our evaluation function:

```
> cata exprEval (runExpr (1 + 2 ∗ 3))
6
```

5. Error algebras

The catamorphisms above do not take the possibility of failure into account, nor do they use position information. In this section, we show how to make use of position information to provide better error messages for algebras supporting failure. We change our evaluation algebra to return an Either (Bounds, String) Int, encoding failure as a *Left* with the position of the error and an informative message, and success with a *Right* value. Furthermore, instead of working on $\mathsf{Expr_F}$, we have the algebra accept Ann Bounds $\mathsf{Expr_F}$ (which is a Functor because $\mathsf{Expr_F}$ is a Functor) so that we have position information available. The new algebra looks as follows:

```
exprEval′ :: Algebra (Ann Bounds Expr_F)
                     (Either (Bounds, String) Int)
exprEval′ (Ann z expr) = case expr of
  Num n   → Right n
  Add x y → do x′ ← x; y′ ← y; return (x′ + y′)
  Sub x y → do x′ ← x; y′ ← y; return (x′ − y′)
  Mul x y → do x′ ← x; y′ ← y; return (x′ ∗ y′)
  Div x y →
    do x′ ← x; y′ ← y;
       if y′ ≡ 0 then Left (z, "division by zero")
                 else Right (x′ `div` y′)
```

Unfortunately, this algebra has to be written in monadic style to propagate the errors. Also, the algebra has to pattern match on the *Ann* constructor to use or discard the position information.

We can improve on this by making the possibility of failure explicit in the algebra type. We introduce a new type of algebra, called an *error algebra*:

```
type ErrorAlgebra f e a = f a → Either e a
```

The major difference between an ErrorAlgebra f e a and an Algebra f (Either e a) is that an ErrorAlgebra has an f a on the left-hand side of the function arrow instead of an f (Either e a). In this way, it assumes that when the catamorphism is applied to the children it is successful and produces a's instead of error values. This means that it is no longer necessary to use monadic style in the algebras:

```
exprEval :: ErrorAlgebra Expr_F String Int
exprEval expr = case expr of
  Num n   → Right n
  Add x y → Right (x + y)
  Sub x y → Right (x − y)
  Mul x y → Right (x ∗ y)
  Div x y | y ≡ 0     → Left "division by zero"
          | otherwise → Right (x `div` y)
```

Whenever a node in the tree produces an error, it no longer fulfills its parent's assumption that it produces an a. The catamorphism function will therefore have to propagate the error upwards to the root of the tree.

There are situations where several children simultaneously produce errors. Rather than arbitrarily picking one of the errors, we *mappend* them together, introducing a Monoid constraint on the error type e. We cannot yet give error algebras to our generic *cata* function, since it expects normal algebras. Also, the applicative computations have not simply disappeared: they need to be applied outside the algebra.

McBride and Paterson (2008) show how to generically capture applicative computations over functors using the type class Traversable:

```
class Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

The instance of Traversable for $\mathsf{Expr_F}$, for example, is as follows:

```
instance Traversable Expr_F where
  traverse f expr = case expr of
    Num n   → pure (Num n)
    Add x y → Add <$> f x ⊛ f y
    Sub x y → Sub <$> f x ⊛ f y
    Mul x y → Mul <$> f x ⊛ f y
    Div x y → Div <$> f x ⊛ f y
```

By capturing $\mathsf{Expr_F}$'s traversal in a generic function, it can be reused in different circumstances, including our error algebras. By adding a Traversable constraint to our functors, we can convert any error algebra into a normal one, collecting errors as we go and producing an algebra we can supply to *cata*:

```
cascade :: (Traversable f, Monoid e)
          ⇒ ErrorAlgebra f e a → Algebra f (Except e a)
cascade alg expr = case sequenceA expr of
  Failed xs → Failed xs
  OK tree′ → case alg tree′ of
    Left xs   → Failed xs
    Right res → OK res
```

The Except datatype we use above is also described by McBride and Paterson. It is similar to Either, but it is designed to be used only in an applicative way so that sequencing two errors results in the combination of those errors (using *mappend*). The monadic Either, on the other hand, discards any errors other than the first. In this way, Except provides the collecting behavior we described before:

```
data Except e a = Failed e | OK a

instance Monoid e ⇒ Applicative (Except e) where
  pure                       = OK
  OK f      ⊛ OK x           = OK (f x)
  OK _      ⊛ Failed e       = Failed e
  Failed e  ⊛ OK _           = Failed e
  Failed e1 ⊛ Failed e2      = Failed (e1 `mappend` e2)
```

Although we now have convenient algebras with error functionality, we do not yet make use of potential annotations. To do something useful with the annotations, we need a new catamorphism function, that works on AnnFixs instead of normal Fixs. If we have this new function take error algebras too, we can automatically couple potential errors with the annotations at the positions at which the errors arise, regaining all the functionality that *exprEval′* above has:

```
errorCata :: Traversable f ⇒ ErrorAlgebra f e a
           → AnnFix x f → Except [(e, x)] a
errorCata alg (In (Ann x expr)) =
  case traverse (errorCata alg) expr of
    Failed xs    → Failed xs
    OK     expr′ → case alg expr′ of
                     Left  x′ → Failed [(x′, x)]
                     Right v  → OK v
```

Position information is now automatically coupled to the errors and can be shown to the user for improved error messages.

6. Parsing annotated values

Now that we have adapted consumers for annotated datatypes, we describe how to adapt producers so that annotations are automatically filled-in.

There are many kinds of producers, but we will focus on parsers. There are several parsing libraries in Haskell, but, for our purposes, it does not matter much which one we pick. We will use Parsec (Leijen and Meijer 2001) in our description.

6.1 A parser for $\mathsf{Expr}_{\mathsf{Bare}}$

To properly compare the parsers for annotated and unannotated datatypes, we first show a standard parser for $\mathsf{Expr}_{\mathsf{Bare}}$. Our expression producer consists of two phases: the lexer converting characters to expression tokens, and the actual parser converting these tokens to expression trees. Our token type is:

```
data ExprToken = TNum Int
               | TPlus | TMinus | TStar | TSlash
               | TPOpen | TPClose | TSpace String

isSpace (TSpace _) = True
isSpace _          = False

isNum (TNum _) = True
isNum _        = False
```

The lexer producing these constructors is unimportant, so we omit it. We focus on the parser instead:

```
pToken  = satisfy ∘ (≡)
pExpr   = chainl1 pTerm    (   Add <$ pToken TPlus
                           <|> Sub <$ pToken TMinus)
pTerm   = chainl1 pFactor  (   Mul <$ pToken TStar
                           <|> Div <$ pToken TSlash)
pFactor =     pNum
          <|> pToken TPOpen *> pExpr <* pToken TPClose
pNum    = (λ(TNum n) → Num n) <$> satisfy isNum
```

There is nothing surprising in this code: we show it only for comparative purposes.

6.2 Keeping track of position

Now that we have seen the parser for $\mathsf{Expr}_{\mathsf{Bare}}$, we build one for $\mathsf{Expr}_{\mathsf{Pos}}$ and compare the two. The new parser uses the constructors of $\mathsf{Expr}_{\mathsf{F}}$ (which, in this paper, have the same names as the constructors of $\mathsf{Expr}_{\mathsf{Bare}}$), but insert position information at every level.

To properly annotate the constructed values, the parser needs to keep track of the current position in the input. Parsec provides support for this in the form of line-column information, but our datatype Bounds requires to keep track of ranges of whitespace. Therefore, it makes sense to use a range as our position information. Parsec lets us maintain a user state: in $\mathsf{Parsec_T}$ s u m a, the type variables stand for stream type, user state, underlying monad and result type, respectively. We set u to Range, as this is the state we want to keep track of.

We couple each token with its Bounds. Computing the proper bounds for each token needs to be done before discarding the whitespace tokens from the lexer's output. We combine discarding these tokens and computing the bounds in a single operation:

```
collapse :: Symbol s ⇒ (s → Bool) → [s] → [(s, Bounds)]
collapse isSpace ts = collapse′ (0, symbolSize ls) isSpace r
  where (ls, r) = span isSpace ts

collapse′ :: Symbol s
          ⇒ Range → (s → Bool) → [s] → [(s, Bounds)]
collapse′ _    _       []       = []
collapse′ left isSpace (t : ts) = new : collapse′ right isSpace r
  where (_, leftInner) = left
        rightInner     = leftInner  + symbolSize t
        rightOuter     = rightInner + symbolSize rights
        right          = (rightInner, rightOuter)
        (rights, r)    = span isSpace ts
        new            = (t, Bounds left right)
```

Most of the work is done in *collapse′*. Its first argument is the current offset in the stream (the left margin of the bounds of the next token). The right margins, which become the left margins in the recursive call, are computed from symbol sizes using the Symbol type class (which we explain below). Function *collapse* (the one exposed to the user) does not need an offset because it assumes it is at the start of the input. Apart from the input stream, its only other argument is a predicate that tells which symbols are to be discarded (*isSpace*).

Rather than building *collapse* specifically for ExprTokens, we determine what properties of tokens we need and capture these in a type class:

```
class Symbol s where
  unparse    :: s → String
```

```
symbolSize :: s → Int
symbolSize = length ∘ unparse
```

The first function, *unparse*, converts a symbol back to a String, exactly the way it was encountered during parsing. The second function, *symbolSize*, is used for computing the length of a symbol when printed.

In the definition of *collapse*, *symbolSize* is called on both single symbols and lists of symbols. This is possible because we provide an instance for lists:

```
instance Symbol s ⇒ Symbol [s] where
  unparse    = concatMap unparse
  symbolSize = sum ∘ map symbolSize
```

The new parser will have Range values as user state and will consume tokens coupled with their position information. This is reflected in a type synonym for our parser type:

```
type P s = ParsecT [(s, Bounds)] Range
```

Every time a new token is consumed, the state needs to be updated. We can hide this in the new *satisfy*, which we define using a primitive parser *tokenPrim*:

```
tokenPrim ::   Stream s m t
            ⇒ (t → String)
            → (SourcePos → t → s → SourcePos)
            → (t → Maybe a) → ParsecT s u m a
```

The first argument is a pretty-printing function for the symbol types; we can use *unparse* from our Symbol class here. The second argument tells how to update the source position; we will use our position information for this. The third argument is the predicate passed to *satisfy* telling which token is expected. Our implementation of *satisfy* becomes:

```
satisfy :: (Monad m, Symbol s) ⇒ (s → Bool) → P s m s
satisfy ok =
  do let pos _ (_, bounds) _ =
           newPos "" 0 (fst (rightMargin bounds) + 1)
         match x@(tok, _) | ok tok    = Just x
                          | otherwise = Nothing
     (tok, nBounds) ← tokenPrim (unparse ∘ fst) pos match
     setState (rightMargin nBounds)
     return tok
```

We use Parsec's *newPos* to create a position based on our margins.

Getting the current position in the stream is now simply getting the current user state:

```
getPos :: Monad m ⇒ P s m Range
getPos = getState
```

6.3 Building recursively annotated values

Now that we maintain obtain position information in the parser state, we construct $\mathsf{Expr}_{\mathsf{Pos}}$ values. We start with the base case: number literals. In $\mathsf{Expr}_{\mathsf{F}}$, the constructor for number literals has type:

```
Num :: Int → ExprF r
```

This constructor, when applied to a number, has a type that matches the type of AnnFix_1 Bounds $\mathsf{Expr}_{\mathsf{F}}$. We just need to wrap the Bounds around it using *mkAnnFix*. We ask for the left margin before parsing the literal token, and the right margin afterwards:

```
type ExprParser = P ExprToken Identity

pNum :: ExprParser (AnnFix Bounds ExprF)
pNum = unit $ (λ(TNum n) → Num n) <$> satisfy isNum
```

```
unit   :: Monad m ⇒ P s m (AnnFix1 Bounds f)
         → P s m (AnnFix Bounds f)
unit p = do left ← getPos
            x    ← p
            mkBounded left x

mkBounded :: Monad m ⇒ Range → AnnFix1 Bounds f
              → P s m (AnnFix Bounds f)
mkBounded left x =
  do right ← getPos
     return (mkAnnFix (Bounds left right) x)
```

We define *pNum* using a number of auxiliary functions which are useful for the other parsers as well. We introduce a type synonym ExprParser for our expression parser: it consumes ExprTokens and uses an underlying Identity monad. The implementation of *pNum* is equal to that of the unannotated parser except for the call to *unit*. Function *unit* takes a parser that yields a value annotated everywhere except at the top level, and turns it into a parser that yields a fully annotated value, by wrapping the current position information around it. This is a useful combinator for parsers that produce simple nodes (such as number literals).

For parsers that do not produce simple nodes, it is often the case that the call to retrieve the right margin is followed by a call to *mkAnnFix*. This is reflected in function *mkBounded*, which, when given the left margin, sets the right margin as the current position, building an AnnFix.

In the parser for $\mathsf{Expr}_{\mathsf{Bare}}$, the branches of the expression trees were built using *chainl1*. For annotated expressions, we need to adapt *chainl1* so that it takes into account annotated and unannotated expressions:

```
chainl1 :: Monad m
         ⇒ P s m (AnnFix Bounds f)
         → P s m (   AnnFix  Bounds f → AnnFix Bounds f
                  → AnnFix1 Bounds f)
         → P s m (AnnFix Bounds f)
```

In our *chainl1*, the first argument is again the parser for the operands, and the second is the parser for the binary operator. Let us consider our expression operator $Add :: \mathsf{r} \to \mathsf{r} \to \mathsf{Expr}_{\mathsf{F}}\ \mathsf{r}$. If we give annotated children to *Add*, we get an AnnFix_1, which is reflected in *chainl1*'s type. It is *chainl1*'s task to insert the right position information.

```
chainl1 px pf = do left ← getPos
                   px >>= rest left
  where rest left = fix $ λloop x → option x $
                      do f ← pf
                         y ← px
                         mkBounded left (f x y) >>= loop
```

We can now define the parser for $\mathsf{Expr}_{\mathsf{Pos}}$. Using our redefined parser combinators, nothing in the parser for $\mathsf{Expr}_{\mathsf{Bare}}$ needs to change (apart from *pNum*). The code is syntactically identical to the previous implementation:

```
pExpr   :: ExprParser (AnnFix Bounds ExprF)
pExpr   =  chainl1 pTerm     (  Add <$ pToken TPlus
                            <|> Sub <$ pToken TMinus)
pTerm   :: ExprParser (AnnFix Bounds ExprF)
pTerm   =  chainl1 pFactor   (  Mul <$ pToken TStar
                            <|> Div <$ pToken TSlash)
pFactor :: ExprParser (AnnFix Bounds ExprF)
pFactor =  pNum
       <|> pToken TOpen *> pExpr <* pToken TClose
```

We have shown a small example, and left out numerous other parser combinators. However, all of these can be adapted to position-saving variants. In our library we provide also *chainr1*, for example.

7. Exploring annotated trees

7.1 Representing structural tree selections

Given an annotated subtree of type AnnFix x f, we can find the corresponding text selection simply by extracting the *Bounds* value in the *Ann* constructor. To convert in the other direction, we search the tree for a node whose bounds match the text selection. In this section we will introduce functions for that purpose.

There are various choices for the result of an operation from a text selection to a tree. The result could be just the selected subtree, but then the context of that subtree is lost. A subsequent edit operation will require the entire tree as input again. One way to solve this is to return a path from the root to the subtree instead. Such a path can be modeled as a list of child indices [Int]. This gives enough context, but it is poorly typed: a path can apply to any tree, and there is no guarantee that a path is valid for a given tree.

The traditional, functional way for representing structural navigation is the zipper (Huet 1997). A zipper datatype is derived from another datatype: its definition depends on the shape of the original datatype. McBride (2001) shows how to automatically derive the zipper for any datatype.

A value of a zipper type represents a particular node in a tree together with the rest of the tree. The selected node is called the zipper's focus, whereas the rest of the tree is called the context. A zipper enables navigation in the tree, stepping from one node to its sibling, child or parent in $O(1)$ time. The zipper also allows the current focus to be updated in $O(1)$ time.

So far, we have been generic over the particular shape functor f, albeit under some class constraints to have access to certain functions. It is hard to encode a zipper in such a setting because we cannot automatically derive the zipper datatype. For this section, we will use a simplified zipper-like structure for navigation on a tree, but we will not have $O(1)$ time updates for the focus. In Section 8.5 we present a more general solution using a real zipper. For now, our zipper data structure is:

```
data Zipper a = Zipper { zFocus :: a
                       , zUp    :: Maybe (Zipper a)
                       , zLeft  :: Maybe (Zipper a)
                       , zRight :: Maybe (Zipper a)
                       , zDown  :: Maybe (Zipper a) }
```

This datatype can be used for tree selections of any tree, not just those expressed in open recursion style. However, we will only construct and use zipper values of type Functor f ⇒ Zipper (Fix f). Despite lacking constant-time updates, this Zipper is still usable as a representation of structural selections. To build a zipper value, we will need the *fold* function from the Foldable class:

```
fold :: (Foldable t, Monoid m) ⇒ t m → m
```

This function says that every Foldable can be seen as a container of elements, and these elements can be visited and combined using *mappend* and *mempty*. If the list monoid [a] is chosen, the result is a list with all the elements in the container. This is exactly what the standard function *toList* :: Foldable t ⇒ t a → [a] does.

For our fixpoint functors, f's elements are its children, and therefore we can use *toList* to obtain the functor's children. We use *toList* in our definition of *enter*:

```
enter :: Foldable f ⇒ Fix f → Zipper (Fix f)
enter f = fromJust (enter' Nothing Nothing [f])
enter' ::   Foldable f
         ⇒ Maybe (Zipper (Fix f)) → Maybe (Zipper (Fix f))
         → [Fix f] → Maybe (Zipper (Fix f))
enter' _  _    []                  = Nothing
enter' up left (focus@(In f) : fs) = here where
  here  = Just (Zipper focus up left right down)
  right = enter' up here fs
  down  = enter' here Nothing (toList f)
```

The helper function *enter'* is given more context: its first argument is the parent (if there is one) of the node that will be produced, and its second argument is its left sibling (if it exists). The third argument is the list of the resulting node's right siblings, still to be processed. In the **where** clause we build the current focus and the recursive values, ensuring optimal sharing.

Once the zipper structure is created, it can be navigated using the record selectors *zDown*, *zUp*, *zLeft* and *zRight*. From anywhere in the zipper, we can recover the original tree by traversing up as far as possible and then requesting the focus:

```
leave :: Zipper a → a
leave z = maybe (zFocus z) leave (zUp z)
```

Navigating down always selects the first child. A useful helper function is navigating down into the *n*th child. Like all other traversal functions, it might fail, so its result is wrapped in a Maybe:

```
child :: Int → Zipper a → Maybe (Zipper a)
child 0 = zDown
child n = child (n − 1) >=> zRight
```

Since Maybe is a Monad, navigation functions like *child* and those of the *Zipper* constructor can be easily composed using the Kleisli arrow composition operator (>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c).

We will now see how we use this zipper to traverse annotated trees.

7.2 Annotation-guided exploring

We define functions from textual selections to a zipper. A naive implementation would visit the entire tree, starting at the root, and at each node searching recursively down and to the right until a node whose bounds match the query range is found. But the laws outlined in Section 2 give us more information, allowing us to prune entire subtrees in some cases:

- If the left offset of the query range is strictly lower than the current node's inner right offset, we do not have to look at the node's right siblings, since law 2 states that children appear in order and their inner ranges do not overlap.
- If the query range is not contained within the current node's outer range, we do not have to consider the node's children anymore, by law 3.

Generalizing these choices for arbitrary annotations x, we can encode the choices using a function of type x → ExploreHints, where ExploreHints is defined as follows:

```
data ExploreHints = ExploreHints { matchHere    :: Bool
                                 , exploreDown  :: Bool
                                 , exploreRight :: Bool }
```

Although uncommon, a parse tree may be constructed in such a way that a parent and its single child have the exact same bounds. For this reason, we return the full list of matching tree selections. Our complete exploration function follows:

```
explore :: Foldable f ⇒ (x → ExploreHints) → AnnFix x f
        → [Zipper (AnnFix x f)]
explore hints = explore' hints ∘ enter
```

```
explore' :: Foldable f ⇒ (x → ExploreHints)
         → Zipper (AnnFix x f) → [Zipper (AnnFix x f)]
explore' hints root =
  [z | (dirOk, zs) ← dirs, dirOk (hints x), z ← zs] where
    In (Ann x _) = zFocus root
    dirs         =
      [(matchHere,    [root])
      ,(exploreDown, exploreMore (zDown root))
      ,(exploreRight, exploreMore (zRight root))]
    exploreMore = maybe [] (explore' hints)
```

The actual work is delegated to *explore′* which takes a zipper as input. It is easier to work on zippers because they allow abstraction over the navigation of the tree. The **do**-block is written in the list monad, exploring the tree recursively in three relevant directions: first the current node, then down, and finally to the right, but only if the hints allow so.

Now we can express our positional conversion function in terms of *explore*:

```
selectByRange :: Foldable f ⇒ Range → AnnFix Bounds f
              → Maybe (Zipper (AnnFix Bounds f))
selectByRange range@(left, _) =
  listToMaybe ∘ reverse ∘ explore hints where
    hints bs@(Bounds _ (ir, _)) = ExploreHints {
        matchHere    = range `rangeInBounds` bs
       ,exploreDown  = range `rangeInRange` outerRange bs
       ,exploreRight = left ⩾ ir}
```

explore returns the topmost matching node, so *selectRange* reverses the returned list and wraps the first result in a *Just*.

Other conversion functions are possible, such as selecting a single position instead of a range. We provide those in the accompanying library.

7.3 Repairing and navigating text selections

The last two use cases we will see are the repair of invalid text selections and navigation based on text selections.

We can distinguish between invalid and valid text selections: a text selection is valid with respect to a parse tree if it corresponds to a structural selection in this parse tree. However, invalid selections are not useless: we can make a good estimate as to what piece of text the user intended to select, based on the erroneous text selection and the list of all the text selections that would have been valid. We do this in function *repairBy*:

```
repairBy :: (Foldable f, Ord d) ⇒ (Range → Range → d)
         → AnnFix Bounds f → Range → Bounds
repairBy cost tree range =
  head (sortOn (cost range ∘ innerRange) (validBounds tree))

sortOn :: Ord b ⇒ (a → b) → [a] → [a]
sortOn = sortBy ∘ comparing

validBounds :: Foldable f ⇒ AnnFix Bounds f → [Bounds]
validBounds (In (Ann b f)) =
  b : concatMap validBounds (toList f)
```

Function *repairBy* takes a tree and a text selection. It asks for all the selections that would have been valid using *validBounds* and sorts them according to some cost function, to which the inner bounds are given. For this we use *sortOn*, which sorts a list based on a property of all elements in the list. The first element of the resulting, sorted list is returned. Using *head* here is safe because the list is guaranteed to contain at least one element: the bounds of the root of the tree.

One possible cost function is *distRange*, which takes the sum of the absolute differences of two ranges' endpoints. Function *repair* is *repairBy* specialized to this particular cost function:

```
repair :: Foldable f ⇒ AnnFix Bounds f → Range → Bounds
repair = repairBy distRange

distRange :: Range → Range → Int
distRange (l1, r1) (l2, r2) = abs (l1 − l2) + abs (r1 − r2)
```

Finally we tackle the issue of navigation based on text selections. Suppose a user selects a piece of text that corresponds precisely to a structural selection. Now the user wants the selection to expand to the direct parent of the selected node, for instance.

We can accomplish this by translating the text selection to a zipper, moving up in the zipper and then translating back to text selection. We can capture these actions using our zipper arguments. All four selectors of the *Zipper* constructor have the type Zipper a → Maybe (Zipper a). From this type we can see that the type of the composition of two movements, e.g. *zDown* >=> *zRight*, is also Zipper a → Maybe (Zipper a).

The index of the zipper is always polymorphic in such functions. We can express this by encoding movements in a **newtype**:

```
newtype Nav = Nav {nav :: ∀a.Zipper a → Maybe (Zipper a)}
```

which is used in a function for navigation based on text selections as follows:

```
moveSelection :: Foldable f ⇒ AnnFix Bounds f
              → Nav → Range → Maybe Bounds
moveSelection tree (Nav nav) range =
  (rootAnn ∘ zFocus) <$> (selectByRange range tree >>= nav)

rootAnn :: AnnFix x f → x
rootAnn (In (Ann x _)) = x
```

We choose to return Maybe Range rather than Maybe Bounds, as Bounds contains strictly more information.

Summing up. Using datatypes in open recursion style has a number of benefits. By building the datatypes we require from smaller building blocks ($\mathsf{Expr_F}$, Ann, Fix), we have gained a generic scheme for expressing morphisms, including catamorphisms and error catamorphisms. By adding certain constraints to our trees, such as Traversable, we can convert between normal algebras and error algebras. We can also generically discard annotations and use both normal algebras and error algebras on both normal and annotated trees.

In terms of producers, the parser we built originally for $\mathsf{Expr_{Bare}}$ only needed minimal changes to work with the new annotated expressions, since most of the work is hidden in the combinators. We can also generically express structural exploration functions using a zipper. For trees annotated with position information, this means we are able to convert between text selections and structural selections, as well as fix invalid selections.

8. Annotations in `multirec`

One of the disadvantages of the approach taken so far is that we have to adapt our original $\mathsf{Expr_{Bare}}$ datatype to open recursion style. Our original intention, however, was to develop a solution for adding position information with as few changes as possible to existing code. Additionally, we are now restricted to working with single, regular datatypes. In particular, recursive families of datatypes, common for expressing large ASTs, are out of our reach.

Swierstra (2008) shows how to take the fixpoint of multiple datatypes using coproducts (lifted sums). However, in his solution there is freedom in which datatype to pick at every recursive position. We do not want this freedom, because we want to specify

which exact datatype to recurse into. In this section we will see how to use the `multirec` library for generic programming (Rodriguez Yakushev et al. 2009) to solve this problem. We omit many of the details due to space constraints: the reader is referred to Van Steenbergen (2010) for a detailed description.

So far we have used arithmetic expressions as our running example. To exploit `multirec`'s capabilities of working with families of datatypes, we will modify the example to use an extra datatype:

```
data Expr = Add   Expr Expr
          | Mul   Expr Expr
          | Tup   Expr Expr
          | Num   Int
          | Typed Expr Type
data Type = Int_T
          | Tup_T Type Type
```

We have replaced the constructors for division and subtraction with two other constructors: one for creating tuples of expressions and one for typed expressions. The latter uses a separate type Type to describe types of expressions.

8.1 Pattern functors

The `multirec` library for generic programming in Haskell supports generic programming over systems of (possibly mutually recursive) datatypes. It uses embedding-projection pairs between user-defined datatypes and their functorial views (which are similar to the open recursion style datatypes). Because `multirec` uses higher-order fixpoints in the functorial views, we get higher-order functorial views of datatypes, which allow us to express recursion into other datatypes. Embedding-projection pairs are encapsulated in a type class:[2]

```
class Fam φ where
  from :: ix → PF φ I* ix
  to   :: PF φ I* ix → ix
```

The function *from* translates from values of the original datatype to the generic representation, while *to* does the inverse. There are many technical details of `multirec` that we cannot explain due to space constraints. We will only highlight a few important concepts. The argument to the Fam class is a type representing the family of datatypes. Function *from* takes a value of type ix (which belongs to the family) and returns its representation as a pattern functor.

The pattern functor is an encoding of a datatype using a sum-of-products structure and abstracting over what occurs at the recursive positions. It is implemented as a type family (Schrijvers et al. 2008):

```
type family PF φ :: (⋆ → ⋆) → ⋆ → ⋆
```

Given a type representing a family of datatypes, PF returns the associated representation type. The second argument to PF, of kind $\star \to \star$, controls what happens at the recursive occurrences. This is a crucial argument, as it allows us to specify the shape of the children of a value. The last argument specifies which particular type of the family ϕ we are focusing on.

Because the pattern functor abstracts over the recursive positions, we can use a higher-order fixpoint to encode the recursion for families of datatypes:

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

[2] Throughout our explanation, we elide explicit witness terms in `multirec` (of type ϕ ix) for simplicity. Details about these can be found in the original paper (Rodriguez Yakushev et al. 2009), and in our implementation.

Since we are using fixpoints again, we can insert annotations at every level. We can now build the `multirec` version of AnnFix and AnnFix_1:

```
type AnnFix  x φ = HFix (K x :*: PF φ)
type AnnFix_1 x φ = PF φ (AnnFix x φ)
```

The types K and :∗: are part of `multirec`'s representation types: in AnnFix we couple the pattern functor with a constant type x (the type of annotations). Generally, we instantiate x to Bounds.

8.2 Pattern functors are traversable

After translating a value to its pattern functor, we still need to be able to traverse and change it. For that we use `multirec`'s *hmapA*:

```
hmapA :: Applicative a
      ⇒ (∀xi.r xi → a (r xi)) → f r ix → a (f r ix)
```

The first argument is the action that is applied to every child. Since we do not know in advance the type of the recursive position, this function is polymorphic on its index. The type guarantees that it works for all indices and that the action does not change the index of a specific child. This function is implemented on each of the basic representation types of `multirec`, which means that all pattern functors (which are built using the representation types only) are traversable with *hmapA*.

8.3 Error catamorphisms in MultiRec

The next step is to translate the error catamorphisms we discussed in Section 5 to use pattern functors. It is reasonably easy to express an algebra in terms of pattern functors in the same way as was done with the base functors. However, now we need to take into account the higher-order recursion parameter and the extra index of the pattern functor:

```
type ErrorAlg_PF f e a = ∀ix. f (K* a) ix → Either e a
```

Here we use K_* a for the recursive occurrences, which says that the children should always contain a value of type a, regardless of their index (K_* is the type-level equivalent of *const*). The universal quantification says that it does not matter what the index of the ingoing value is: the result is always Either e a.

With this algebra type we can write the `multirec` version of *errorCata*:

```
errorCata :: ErrorAlg_PF f e r → HFix (K x :*: f) ix
          → Except [(e, x)] r
errorCata alg (HIn (K k :*: f)) =
  case hmapA (λg → K* <$> errorCata alg g) f of
    Failed xs → Failed xs
    OK expr' → case alg expr' of
                  Left x'  → Failed [(x', k)]
                  Right v → OK v
```

This implementation is very similar to that of the earlier *errorCata*. However, writing an algebra for this catamorphism is not as easy as before, since the algebra cannot use the constructors of the original datatype. Instead, it has to pattern match on the representation types, which are far more verbose and make the algebra less clear.

This problem is not specific to our error algebras and already occurs in normal algebras. Rodriguez Yakushev et al. (2009) solve the problem by automatically translating the pattern functor to a convenient algebra type (using a type family). We adapt this strategy also for our error catamorphisms by defining a type family:

```
type family ErrorAlg
  (f :: (⋆ → ⋆) → ⋆ → ⋆)   -- pattern functor
  (e :: ⋆)                 -- error type
```

```
(a :: ⋆)              -- result type
                  :: ⋆ -- resulting algebra type
```

This type is instantiated to each of the representation types, and an associated function *mkErrorAlg* converts regular error algebras to convenient error algebras. We do not show the details of ErrorAlg; instead, we show an example algebra that infers the type of expressions of our language of tuples:

```
inferType ::   ErrorAlg PF_Expr String Type
          :&: ErrorAlg PF_Type String Type
inferType =  (compare True "+" & compare True "*" & tup
                & const (Right Int_T) & compare False "::")
           & (Right Int_T & tup) where
  compare b op ty1 ty2
    | ty1 ≡ ty2  = Right ty1
    | otherwise =
        let text = if b
                   then "lhs and rhs should " ++
                        "have equal type"
                   else "lhs is of type rhs"
        in Left ("in lhs" ++ op ++ "rhs, " ++ text)
  tup ty1 ty2 = Right (Tup_T ty1 ty2)
```

This algebra says that the operands of $+$ and $*$ must have equal types. It also checks whether explicit type signatures using :: match the types of the expressions on the left-hand sides. The use of the combinators :&: and & helps to make the algebra more similar to the structure of the original datatype.

To test this algebra, we use a function *readExpr* :: String → AnnFix Bounds Tuples Expr, which we will define in Section 8.4:

```
> let expr1 = readExpr "(1, (2,3))"
> errorCata (mkErrorAlg inferType) expr1
OK (TyTup TyInt (TyTup TyInt TyInt))
> let expr2 = readExpr "(1 :: (Int,Int), 2+(3,4))"
> errorCata (mkErrorAlg inferType) expr2
Failed
  [ ( "in x::t, x should have type t"
    , Bounds { leftMargin  = (1,1)
             , rightMargin = (16,16) } )
  , ( "in x+y, x and y should have equal type"
    , Bounds { leftMargin  = (17,18)
             , rightMargin = (28,28) } ) ]
```

8.4 Constructing recursively annotated trees

In Section 8.1 we have translated the AnnFix and AnnFix_1 type synonyms from Section 3 to use `multirec` concepts. Function *mkAnnFix* can be translated in a similar fashion:

```
mkAnnFix :: x → AnnFix1 x s ix → AnnFix x s ix
mkAnnFix x = HIn ∘ (K x :*:)
```

Previously, we produced AnnFix_1s by supplying fully annotated AnnFixs to a constructor of Expr_F. However, producing an AnnFix_1 in `multirec` is not as easy: we cannot supply annotated children to the constructors of the original datatypes Expr and Type because they only accept unannotated values. Instead, to produce an AnnFix_1 we have to use the pattern functor versions of the constructors. For example, given two annotated expressions of type AnnFix Bounds Tuples Expr, we can construct their sum using constructor *Add*'s pattern functor constructor $\lambda x\ y \rightarrow L \circ Tag \circ R \circ L \$ I\ x :\!*\!:\ I\ y :\!*\!:\ U$. These constructor functions are long and tedious, and should not be exposed to the user of the library.

One possible solution would be to generate smart constructors with convenient names. Van Noort et al. (2008) present a solution that could potentially be adapted to our situation. The approach we take, however, makes use of the fact that the parsers we work with construct trees in post-order form: first the children are parsed and constructed (including position information), then the node itself. For example: to parse the expression $2+3$, first 2 is parsed, then 3, and then the sum is constructed. Ommitting the details about position for simplicity, this parser operates as follows:

```
do n2 ← yield (Bounds ...) (Num 2)
   n3 ← yield (Bounds ...) (Num 3)
   n5 ← yield (Bounds ...) (Add n2 n3)
   return n5
```

Here, *yield* is taking concrete datatypes constructors, such as *Add*, which are built using other concrete datatypes (returned by *yield*). However, the position information cannot be stored in these constructors. What *yield* does is to check the number of children of its last argument. If there are no children, like for *Num*, the constructed value (using the generic representation constructors) is put on a stack. If the constructor expects children, like *Add*, the correct number of elements is popped from the stack and used to replace the current children of the term. Thus, in the example above, supplying *Add* ⊥ ⊥ to *yield* would have the same effect.

In case there are not enough children in the stack, or they have the wrong type, a runtime error is returned. This solution is not ideal, because it cannot be applied to all types or parsers. Additionally, mistakes made in the parser are harder to find. However, it allows us to use the original constructors of the datatype.

```
data FixZipper φ f ix
enter :: Zipper φ f ⇒ HFix f ix → FixZipper φ f ix
leave :: Zipper φ f ⇒ FixZipper φ f ix → HFix f ix
type Nav = ∀φ f ix.Zipper φ f ⇒ FixZipper φ f ix
                               → Maybe (FixZipper φ f ix)
up, down, left, right :: Nav
on   :: (∀xi.HFix f xi → a) → FixZipper φ f ix → a
```

Figure 2. Interface of the generic zipper.

8.5 Annotation-guided exploration

In contrast with the annotation-guided exploration of Section 3, pattern functors give us enough information to define genuine zippers, as described by Rodriguez Yakushev et al. (2009). However, since our annotations change the structure of the representation, we cannot use the standard zipper. Instead, we slightly generalize the zipper to behave adequately on annotated representations. We present only the interface of our zipper, not its implementation.

An overview of the interface of the generic zipper is given in figure 2. The names of the arguments of FixZipper are consistent with what we have seen before: ϕ is the type representing the family of types, f is the pattern functor over which the zipper is computed and ix is the top-level index. Function *enter* takes a fixpoint and returns a zipper over that fixpoint, with the focus at the tree's root. Function *leave* leaves the zipper structure by navigating to the top and returning the tree. The navigation functions *up*, *down*, *left*, and *right* all have the same type, which, as before, is called Nav. Navigation steps can be composed using >=>. Finally, *on* takes a function and applies it to the current focus.

If we instantiate a FixZipper with K x :*: PF ϕ for f, it holds selections of recursively annotated trees:

```
type AnnZipper φ x = FixZipper φ (K x :*: PF φ)
```

Given a zipper over an annotated tree, we can extract the annotation of the current focus using *on*:

$$
\begin{aligned}
&\mathit{focusAnn} :: \mathsf{AnnZipper}\ \phi\ \mathsf{x}\ \mathsf{ix} \to \mathsf{x} \\
&\mathit{focusAnn} = \mathit{on}\ (\lambda(\mathit{HIn}\ (K\ x :*: _)) \to x)
\end{aligned}
$$

Function *explore* plays a central role in Section 7. Most other functions are expressed in terms of *explore*. Its counterpart in `multirec` is very similar:

$$
\begin{aligned}
&\mathit{explore} :: \mathsf{Zipper}\ \phi\ (\mathsf{PF}\ \phi) \Rightarrow (\mathsf{x} \to \mathsf{ExploreHints}) \\
&\qquad \to (\mathsf{AnnFix}\ \mathsf{x}\ \phi)\ \mathsf{ix} \to [\mathsf{AnnZipper}\ \phi\ \mathsf{x}\ \mathsf{ix}] \\
&\mathit{explore}\ \mathit{hints} = \mathit{explore}'\ \mathit{hints} \circ \mathit{enter} \\
&\mathit{explore}' :: \mathsf{Zipper}\ \phi\ (\mathsf{PF}\ \phi) \Rightarrow (\mathsf{x} \to \mathsf{ExploreHints}) \\
&\qquad \to \mathsf{AnnZipper}\ \phi\ \mathsf{x}\ \mathsf{ix} \to [\mathsf{AnnZipper}\ \phi\ \mathsf{x}\ \mathsf{ix}] \\
&\mathit{explore}'\ \mathit{hints}\ \mathit{root} = \\
&\quad [z \mid (\mathit{dirOk}, \mathit{zs}) \leftarrow \mathit{dirs}, \mathit{dirOk}\ (\mathit{hints}\ x), z \leftarrow \mathit{zs}]\ \textbf{where} \\
&\qquad x \quad = \mathit{focusAnn}\ \mathit{root} \\
&\qquad \mathit{dirs} = [(\mathit{matchHere}, \quad [\mathit{root}]) \\
&\qquad\qquad ,(\mathit{exploreDown}, \mathit{exploreMore}\ (\mathit{down}\ \mathit{root})) \\
&\qquad\qquad ,(\mathit{exploreRight}, \mathit{exploreMore}\ (\mathit{right}\ \ \mathit{root}))] \\
&\qquad \mathit{exploreMore} = \mathit{maybe}\ [\,]\ (\mathit{explore}'\ \mathit{hints})
\end{aligned}
$$

The type of function *explore* is slightly more complicated, but its behavior remains the same. The annotation is extracted using *focusAnn*, rather than by means of pattern matching.

The other generic zipper functions in `multirec`, including *selectByRange* and *repairBy*, are also very similar to their counterparts in Section 7.

9. Related work

9.1 `GroteTrap`

A library related to the one we present is `GroteTrap` (Leeuwestein and Steenbergen 2008). Its purpose is to provide an easy way to define expression languages and get functions for converting between text selections and structural selections for free.

As an example, we show how we could express the language of $\mathsf{Expr}_{\mathsf{Bare}}$ of Section 1 in `GroteTrap`:

$$
\begin{aligned}
&\mathit{exprLanguage} :: \mathsf{Language}\ \mathsf{Expr}_{\mathsf{Bare}} \\
&\mathit{exprLanguage} = \mathit{language} \\
&\quad \{\mathit{number} \quad = \mathit{Num} \\
&\quad ,\mathit{operators} = [\mathit{Assoc}\ \mathit{Add}\ 1\ \texttt{"+"}, \mathit{Assoc}\ \mathit{Sub}\ 1\ \texttt{"-"} \\
&\qquad\qquad ,\mathit{Assoc}\ \mathit{Mul}\ 2\ \texttt{"*"}, \mathit{Assoc}\ \mathit{Div}\ 2\ \texttt{"/"}]\}
\end{aligned}
$$

The language definition ties the constructors to lexical constructs, providing enough information for the generation of a parser. Since the available constructs are limited to numbers, variables and unary and binary operators, parsed expressions can be stored in a universal datatype with one constructor for each type of construct. Coupling this with position information allows for conversion between textual and structural selections.

Although `GroteTrap` works well for small expression languages, any grammar that requires more than just identifiers, numbers, and unary and binary operators is very hard if not impossible to define. The library we present makes the selection conversion functions automatically available for any context-free grammar that can be represented by a datatype that `multirec` can deal with.

9.2 Proxima

Proxima (Schrage 2004) is a generic framework for creating structure editors. Programmers have full control over the presentation of datatype values, and presentation level edit operations are mapped back to the data model. Proxima provides tools for defining parsers, ASTs, and evaluation using attribute grammars. It also supports selections in both textual and structural views. However, since it uses a universal datatype to represent all datatypes, the problem only has to be solved for this datatype. In contrast, our solution is datatype-generic.

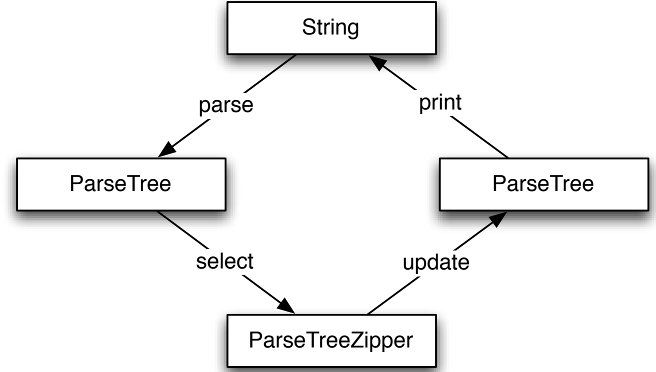

Figure 3. The refactoring cycle.

10. Future work

10.1 Dealing with tree updates

Figure 3 depicts the refactoring cycle: in an Integrated Development Environment (IDE), a programmer selects a source code fragment and performs a refactoring action. To support this, the IDE maps the text selection to a structural selection in the syntax tree, something we have discussed in detail in the previous chapters. Then the selection is processed so that the refactoring is carried out, changing the tree in the process. Perhaps part of the code is deleted or moved to a new position, or new code is created.

Dealing with changes to the tree is an interesting problem that we have not addressed yet. In particular, we have to make sure that the text positions stored in the annotated tree stay correct after an edit. A possible solution is to not store position information but the exact source code responsible for each subtree. From this, position information can be inferred.

Doing this would introduce a whole new set of questions: can we generically add source code to trees? Do we need to create a derived datatype for this? How can we efficiently compute position information from the source code?

10.2 Contexts in algebras

To use annotated trees, our approach requires that a programmer expresses data-consuming functions as algebras for catamorphisms. Most consumers can be expressed in terms of an algebra, because the result type of an algebra is allowed to be a function of type $\mathsf{c} \to \mathsf{r}$, where c can be any context information necessary to compute the result. A well-known example for which we have to add a context is the evaluation of arithmetic expressions with variables. The result type of the algebra is something like $\mathsf{Environment} \to \mathsf{Int}$. When evaluating a variable, the variable is looked up in the environment to retrieve its value. In the evaluation of a binding, a key-value pair is added to the environment.

Result types that use a context do not work well in combination with error algebras: our type $\mathsf{ErrorAlgebra}\ \mathsf{f}\ \mathsf{e}\ (\mathsf{c} \to \mathsf{a})$ is equivalent to $\mathsf{f}\ (\mathsf{c} \to \mathsf{a}) \to \mathsf{Either}\ \mathsf{e}\ (\mathsf{c} \to \mathsf{a})$. This is not very convenient: the algebra has to decide whether to throw an error without being able to inspect the context. Imagine this in the scenario above: when encountering a variable node, we would like to throw an error if the variable is unbound in the environment. However, we cannot do so because no environment is available. A more convenient algebra type would be $\mathsf{f}\ (\mathsf{c} \to \mathsf{a}) \to \mathsf{c} \to \mathsf{Either}\ \mathsf{e}\ \mathsf{a}$, where the context

is moved out of the Either constructor, and thus available when deciding to return a *Left* or a *Right*. It would be interesting to see if *errorCata* can be changed to work with such algebras.

11. Conclusion

We have shown how to generically add position information to recursive datatypes, along with how to adapt producers and consumers so that the position information is constructed or consumed automatically. Our solution is based on the fixpoint view of datatypes. By expressing a recursive type as a fixpoint, it is possible to insert position information at every recursive position.

We have implemented this solution in two ways: one using datatypes in open recursion style (Section 3), and the other using the `multirec` library for generic programming (Section 8). These implementations are the first steps towards solving the task of adding position information to software tools using a library instead of a design pattern.

Acknowledgments

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant. We thank Andres Löh for his helpful suggestions.

References

Gerard Huet. Functional pearl: The zipper. *JFP*, 7(5):549–554, September 1997.

Jeroen Leeuwestein and Martijn van Steenbergen. GroteTrap. `http://www.haskell.org/haskellwiki/GroteTrap`, June 2008. [Online; accessed 14-June-2010].

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001. Unpublished manuscript, available via `http://strictlypositive.org/diff.pdf`.

Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 18(1):1–13, 2008.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *WGP 2008*, pages 13–24. ACM, 2008.

Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of JFP.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP 2009*, pages 233–244. ACM, 2009.

Martijn M. Schrage. *Proxima—a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, Oct 2004.

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

Martijn van Steenbergen. Generic selections of subexpressions. Master's thesis, Utrecht University, 2010.

Wouter Swierstra. Data types à la carte. *JFP*, 18(4):423–436, 2008.