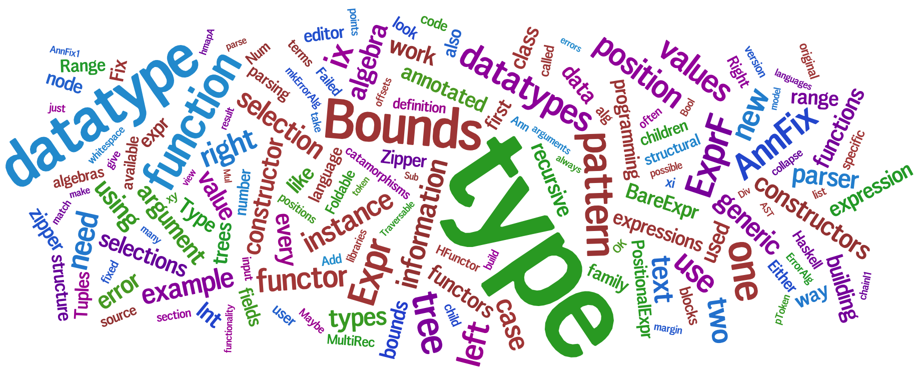
Generic Selections of Subexpressions

Martijn van Steenbergen

MSc Thesis

June 6, 2010

INF/SCR-09-61



Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
prof. dr. J.T. Jeuring
Second Supervisor:
ir. José Pedro Magalhães

Contents

Acknowledgements

A thesis is never written by a single person alone. I've enjoyed a lot of support from various people and I would like to take a moment to thank them:

My supervisors Johan Jeuring and José Pedro Magalhães have always and patiently answered my many questions, offering hints whenever I was stuck and gently guiding me in a good direction.

My sisters, parents and grandparents have continuously supported me through their love and patience.

Most of the work for this thesis was done in the students lab at the Software Technology group at university. Situated conveniently close to the offices of professors, teachers and researchers, I was often able ask them for advice. Notably Andres Lh and Stefan Holdermans have helped me significantly, for example by answering complicated questions about higher-order typery.

Also present in the students lab were other students working on various projects. There was often someone to exchange thoughts with and also to be distracted by. Of these, Chris Eidhof, Erik Hesselink, Jeroen Leeuwestein, Mathijs Swint, Sebastiaan Visser and Tom Lokhorst deserve special mention.

Sjoerd Visscher, my colleague at Q42/Xopus, has provided me with many a helpful hint or thought. An exchange of emails with Conor McBride and my visit to GTTSE in Braga, Portugal have also been insightful.

Thank you!

Chapter 1

Introduction

In the early days of computing, designing and implementing a new programming language was very special. With the many tools that are available to language designers and programmers today, it has become much easier to build a language "just for fun", as seen through the many esoteric programming languages such as LOLCODE (Lin07), Brainfuck (Mül93), Piet (MM01), and Iota and Jot (Bar01).

The increased ease of building a language is partly because of the numerous tools available for building parsers, either through parser combinators such as list of successes (Wad85; HM98), Parsec (LM01) and those in the Utrecht toolset (SAA99) or parser generators such as lex and yacc (LMB92), JavaCC (VS96) and ANTLR (Par07). Another reason is the modularization of compilers: in early days every new language had to build the entire chain of processes that make up a compiler. Nowadays it is common to split a compiler in clearly separated phases that allow reuse. Examples of this are Clojure, Groovy, and Scala that compile to the Java platform and C#, F#, and Visual Basic .NET that compile to the .NET CLI, inheriting all libraries available for those platforms. Another strategy is to compile languages to intermediate languages such as GRIN, LLVM, C--, or even C, for which excellent compilers are already available.

The first steps of designing a new language include the definition of a semantics and a data structure to capture programs. Then follow the parser and algorithms for evaluating programs or compiling them to another language. As a language matures, new features are added or existing ones are improved.

There usually comes a point where the quality of the error messages is improved. This is often implemented later not only because it is very difficult to figure out what the user intended to do (Hee05)—which is necessary to give a good descriptive message—but also because it requires changes that are pervasive throughout the whole compiler. Often extra state needs to be stored and passed around in order to give error messages that make sense. A good example is the location information accompanying error messages: if the user doesn't know where exactly an error occurred, it doesn't matter anymore how good or descriptive the error messages are; the user is not going to accept the compiler as a good one. However, adding support for location information requires changes to various components of the compiler:

- The datatypes that represent a compilation unit need to have fields added for storing the position annotations. Although this change is almost mechanical, it is not a small one: every constructor of every datatype that makes up the model needs to have fields added for this.
- Producers of the now annotated model have to change to fill in the new fields. 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 nodes.
- Consumers of the new model (functions taking the new model as input) need to be adapted as well, either to explicitly ignore the annotations if they are not needed, or to use them in the results if they are.

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.

1.1 Structural selections

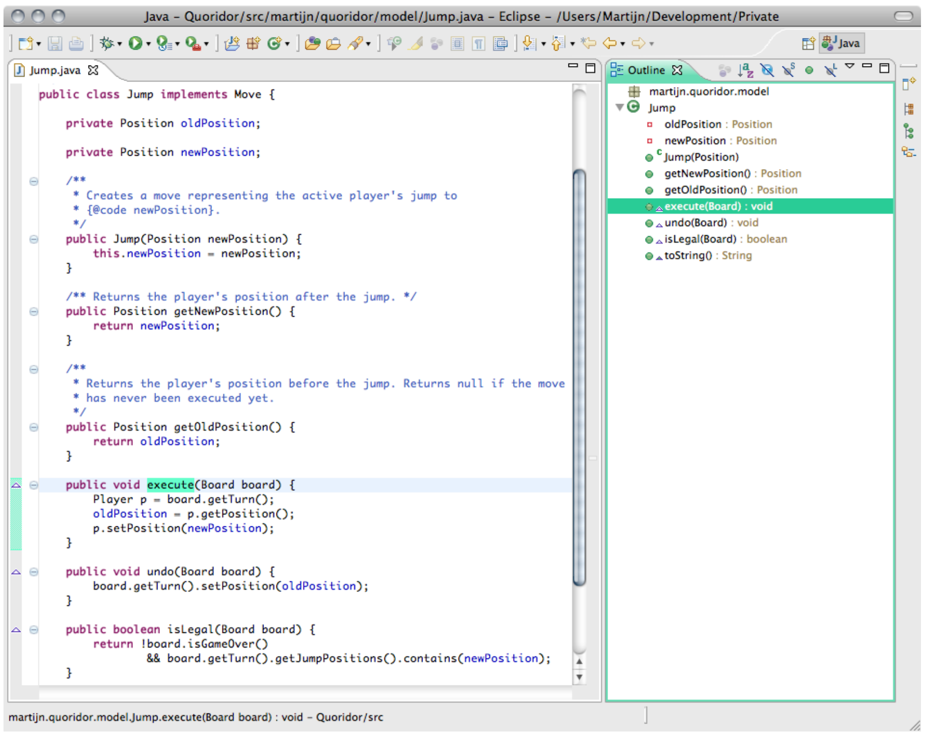
In this thesis the notion of a structural selection comes up often. Let us define more clearly what we mean by structural selections.

Data stored in computers is serialized: linear sequences of bits stored on physical media. Structure emerges when meaning is given to these bits, such that they can be manipulated in useful ways. A sure sign of structure is when an entity is composed of several smaller entities.

Parser technology was developed to take a linear sequence of symbols (bits, bytes, characters, words) and recognize substructures, translating them into an in-memory model where the structure is explicit. This makes it easy for computer programs and programmers to reason about the meaning of the input and manipulate it. Structures and substructures are necessary for introducing variety: how else could you define a formal language with an infinite number of meaningful sentences?

When you have a structure with substructures, it is clear what a structural selection is: it is the selection of one or more particular substructures. Before the structure was unveiled by the parser, all we could do was select ranges of input, such as a text selection in a text editor.

Parsers connect two different worlds of structures. On the one hand they are based on grammars, where the foundations for structure are the production rules that make up the grammar. On the other hand, parsers usually build abstract syntax trees (ASTs). Here the basis for the structure are the constructors that the datatypes of the ASTs are made of.


Java – Quoridor/src/martijn/quoridor/model/Jump.java – Eclipse – /Users/Martijn/Development/Private

```java
public class Jump implements Move {

    private Position oldPosition;

    private Position newPosition;

    /**
     * Creates a move representing the active player's jump to
     * {@code newPosition}.
     */
    public Jump(Position newPosition) {
        this.newPosition = newPosition;
    }

    /** Returns the player's position after the jump. */
    public Position getNewPosition() {
        return newPosition;
    }

    /**
     * Returns the player's position before the jump. Returns null if the move
     * has never been executed yet.
     */
    public Position getOldPosition() {
        return oldPosition;
    }

    public void execute(Board board) {
        Player p = board.getTurn();
        oldPosition = p.getPosition();
        p.setPosition(newPosition);
    }

    public void undo(Board board) {
        board.getTurn().setPosition(oldPosition);
    }

    public boolean isLegal(Board board) {
        return !board.isGameOver()
                && board.getTurn().getJumpPositions().contains(newPosition);
    }
```

```
martijn.quoridor.model
Jump
    oldPosition : Position
    newPosition : Position
    Jump(Position)
    getNewPosition() : Position
    getOldPosition() : Position
    execute(Board) : void
    undo(Board) : void
    isLegal(Board) : boolean
    toString() : String
```

martijn.quoridor.model.Jump.execute(Board board) : void – Quoridor/src

Figure 1.1: In Eclipse, the editor selection and Outline View selection are always in sync.

It is often the case that substructures of an AST directly relate to specific pieces of the source code. It therefore makes sense to ask: what text selection does this structural selection correspond to? The question in the other direction is reasonable, too. A real example of this are the compilers we talked about in the first section. They work on structural ASTs and when they want to refer to a specific selection, they need to express this selection in terms of a text selection in the source code, because that is what the user fed to the compiler and that is what the user manipulates. Another example is the Outline View in the Eclipse editor (Figure 1.1). This view is usually shown next to the source code and provides a hierarchical overview of the source. Positioning the caret in the code highlights the corresponding node in the outline and vice versa: clicking on a node in the view highlights the appropriate text selection in the source. All these functionalities require mapping between the two kinds of selections.

Obviously, there is a lot more freedom in textual selections than in structural selections and not every text selection will correspond to a selection in the AST. Therefore, we distinguish between text selections that do and don't correspond to structural selections. We will call them *valid* and *invalid* selections, respectively.

Many editors work on textual representations of models. Some, however, take a more graphical approach in which the structure is more apparent. In the EQL editor, for example, the user could drag blocks from and to containers.

The EQL editor was built on the Graphical Editing Framework (GEF), which is designed for graphical, structural editors. Another very nice example of a structural editor is the equation editor that comes with Adobe FrameMaker: the user manipulates text but in a structured way in which the tree shape of the expressions is apparent. For example, by pressing the space bar the user can select the next enclosing part of the expression, which corresponds to moving up one level in the AST.

In general purpose programming languages, the distinction between the two structural worlds of grammar production rules and nodes in an AST is always explicit and it is the parser's or pretty printer's job to connect the two. However, some tools specialized for term rewriting such as TXL (Cor06), Stratego (Vis03), and Rascal (KvdSV09) adopt the radical view that *the grammar is the AST*. In this view, the various production rules *are* the constructors of the AST and usually some restrictions are put on the grammar, such as forbidding nested choices (denoted with a vertical bar `|` in BNF) because this introduces anonymous constructors, making it more difficult to work with parse trees later on. Whitespace usually has a special meaning in these languages too, instead of optionally introducing whitespace as something special in the lexer phase. Working with languages in this way offers some important advantages: there is no need to separately specify the AST, parser, and pretty printer anymore, as these are unified into one syntax. Furthermore, rewrite rules can be expressed using the syntax of the language being defined.

This strategy works well in such specialized languages, but is very hard to adapt in general purpose programming languages because they usually don't allow customization of syntax. In general purpose languages, concrete syntax has to be encoded as string literals, making it impossible to achieve strong static types.

1.2 Contributions

This thesis is about making the mapping from text selection to structural selection and vice versa as easy as possible for the developer of a compiler or tool. As many of the issues described in the introduction as possible are hidden from the programmer. The solutions are developed in and for the functional general-purpose programming language Haskell.

As a teaser, here is a code snippet that shows an example use of the API presented in this paper:

```
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)
```

It defines an algebra for evaluating an arithmetic expression. Using this algebra, we may call a function *compileExpr* as follows:

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

Specifically, this thesis makes the following contributions:

- By using fixpoint views, a type-indexed datatype is defined to recursively insert position information in algebraic datatypes (Chapter 5).
- We will see how consumers of datatypes expressed as catamorphisms can be made to work automatically with both the bare datatypes and the derived information-rich datatypes (Section 5.1). Furthermore, a new kind of catamorphism is introduced that makes the possibility of failure explicit, allowing automatic extraction of the relevant position information in case of failure (Section 5.2).
- Several common editor use cases are solved, such as mapping from text selection to structural selection and back and fixing invalid selections (Section 5.4).
- Parser combinators are introduced that help in building recursively annotated ASTs (Section 5.3).
- The solutions to the problems are solved again using the generic programming library MultiRec and compared with the previous solutions (Chapter 6).

Chapter 2

Background

2.1 The EQL editor

The work in this thesis stems from two earlier projects. The first is the development of a graphical editor for a DSL called EQL, developed from 2003-2007 for Hexis B.V. to describe templates for a report generator. Given such a template and data from some source, it is the report generator's job to create a PDF file that presents this data in a pleasant way. In this specific case, the data was assumed to come from a database and so the template files used SQL to indicate what data they were interested in, creating formatted tables and referring to fields from database tables. Report generators have many practical applications. Common ones include managing customer bases, generating e.g. invoices, welcoming letters, reminder letters, and so on. Just as web services dynamically generate documents to be displayed on computers, report generators dynamically generate pages to be printed out.

Although the DSL for the templates was textual and could easily be written by hand, the end result would be something formatted and graphical and so a graphical editor for the templates could offer significant advantages to template designers. This was the purpose of the EQL editor: the user could drag and drop various objects into the editor area.

Even though building a text editor or graphical editor is a lot of work, there are no real inherent difficulties (apart from those originating in the domain): many useful design patterns are applicable, such as the *model-view-controller* pattern and the *command* pattern. The EQL editor, however, was both: it offered both a textual view and a graphical view and the user was free to switch between the two at will. One of the requirements was that changes done in the visual view affected the textual view only locally, leaving the source code unchanged as much as possible and preserving layout and comments. This proved a very interesting challenge.

The solution we chose to solve this challenge was to have the parser not throw away any information at all, preserving every single source token, including whitespace and comments. Changes to the model then always meant changing both the AST and the list of tokens that make up the source code, keeping them synchronized. The solution is explained in detail in *Building a hybrid editor* (vS08).

2.2 Grote Trap

The second related project is *Grote Trap*, a library written by Jeroen Leeuwestein and Martijn van Steenbergen for the 2007 Generic Programming course at Utrecht University, taught by Johan Jeuring. Its purpose was to provide an easy way to define expression languages and get functions for converting between text selections and structural selections for free. This was one of the projects that were available; we chose it because it reminded us of the EQL editor in which similar functionality was available, but specific to the EQL language. We found extending this to work for arbitrary languages an interesting challenge.

As an example, consider the following datatype for expressions of propositional logic:

```
data Logic
  = Var  String
  | Or   [Logic]
  | And  [Logic]
  | Impl Logic Logic
  | Not  Logic
```

By defining an expression of type *Language Logic*, a grammar could be specified and a parser and conversion functions were made available for free. The language definition looked like this:

```
logicLanguage :: Language Logic
logicLanguage = language
  {variable = Just Var
  ,operators =
    [Unary Not Prefix  0 "!"
    ,Assoc And         1 "&&"
    ,Assoc Or          2 "||"
    ,Binary Impl InfixR 3 "->"
    ]
  }
```

The language definition ties the constructors to lexical constructs, providing enough information for the generation of a parser. Because the available constructs were limited to numbers, variables and unary and binary operators, parsed expressions could be stored in a universal datatype with one constructor for each type of construct. Not only were the specific operators stored, the information regarding their positions in the input sentence was also remembered. This allowed converting between textual and structural selections of expressions for which a language definition was provided. Grote Trap is available for download on Hackage. [1]

Although Grote Trap worked well for small expression languages, any grammar that required more than just identifiers, numbers, and unary and binary operators was very hard if not impossible to define. It is from this deficiency that this thesis project arose: the goal of this work is to make the selection conversion functions available automatically for any context-free grammar, however elaborate.

[1] `http://hackage.haskell.org/package/GroteTrap`

Chapter 3

Representing textual selections

Text selections will be used throughout this entire thesis and play an important role. For that reason, we need to decide on a way to represent them. 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.

But behind the scenes programs usually work with offsets, especially if they need to do computations on these offsets. Only at the last moment, right before 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.

Because offsets are easier to work with behind the scenes, we will use offsets to represent text positions in this thesis. A text selection is then a tuple of two offsets. We call the type of text selections *Range*:

$$\textbf{type}\ \mathit{Range} = (\mathit{Int}, \mathit{Int})$$

Offsets can be thought of as positions between two characters. They start at zero. For any text of length n, there are $n + 1$ valid offsets, namely those in the closed interval $[0\,..\,n]$. For ranges $(\mathit{left}, \mathit{right})$ we always maintain the invariant that $0 \leqslant \mathit{left}$ and $\mathit{left} \leqslant \mathit{right}$.

We define some utility functions to work with offsets and ranges:

$$\begin{array}{l}
\mathit{posInRange} :: \mathit{Int} \rightarrow \mathit{Range} \rightarrow \mathit{Bool} \\
\mathit{posInRange}\ \mathit{pos}\ (\mathit{left}, \mathit{right}) = \mathit{left} \leqslant \mathit{pos} \wedge \mathit{pos} \leqslant \mathit{right} \\
\mathit{rangeInRange} :: \mathit{Range} \rightarrow \mathit{Range} \rightarrow \mathit{Bool}
\end{array}$$

$$\begin{aligned} \mathit{rangeInRange}\ (\mathit{left}, \mathit{right})\ \mathit{range} = {} & \mathit{left}\quad \text{`}\mathit{posInRange}\text{`}\ \mathit{range} \\ & \wedge\ \mathit{right}\ \text{`}\mathit{posInRange}\text{`}\ \mathit{range} \end{aligned}$$

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.

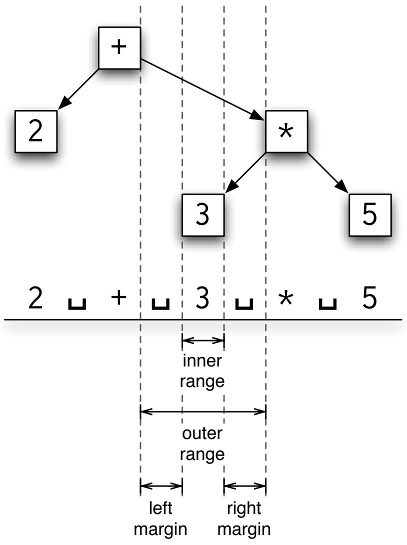


Figure 3.1: The abstract syntax tree for the expression `2 + 3 * 5`. The bounds for the node representing the number 3 are indicated, showing the naming of the various ranges.

To be able to map between text selections and tree selections, we need to remember what each subtree's position in the input was. What information do we need to remember, exactly? To translate from tree selection to text selection, it is sufficient to store a *Range* with every node. However, if a single range is all we have for a node, then to translate a text selection back to a tree selection the user needs to select the *exact* range for the text selection to be recognized. It is often the case that a structure is surrounded by some whitespace in the source text, and 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 turns into actual structural text, the point where the whitespace after the structure starts, and the point where this whitespace ends. Figure 3.1 shows these four offsets for the node representing the 3 in the arithmetic expression `2 + 3 * 5` and the terms we will use for some meaningful combinations of these offsets. Using this information, we can be flexible to the user and accept any text selection that starts between the first two offsets and ends between the last two offsets.

The combination of these four offsets is stored in the datatype Bounds:

```
data Bounds = Bounds {leftMargin :: Range, rightMargin :: Range}
  deriving (Eq, Show)
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. They could have stored 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, because 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. To give an example: the subtree for $3 * 5$ shares its left margin with the node for 3 and its right margin with the node for 5.

We define a function *rangeInBounds* that tells 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
```

The way the parse tree is shown in relation to its source text in figure 3.1 implies some laws we rely on and maintain as invariants when using and constructing annotated parse trees. 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 AST and their inner ranges do not overlap. So 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 a parent's every child we have that $innerRange\ bounds_{child}$ *`rangeInRange`* $innerRange\ bounds_{parent}$.

Chapter 4

Grote Trap extended

As we described in section 1, the first step towards automatic selection conversion is to adapt a datatype to also hold the necessary position information. Early attempts extended Grote Trap's idea and focused on trees of concrete syntax, using a universal tree like this:

```
type ParseTree a = Rose (Bounds, a)
data Rose      a = Rose a [Rose a]
```

Here a parse tree is a standard rose tree with at each node a *Bounds* value and an *a* value, the type of our AST. To understand how *ParseTree* works, let us take a look at an example. We will use the following simple and well-known datatype for arithmetic expressions:

```
data BareExpr
  = Num Int
  | Add  BareExpr BareExpr
  | Sub  BareExpr BareExpr
  | Mul  BareExpr BareExpr
  | Div  BareExpr BareExpr
  deriving (Eq, Show)
```

The parse tree for the sentence `"1 + 2 * 3"` looks like this:

```
example :: ParseTree BareExpr
example =
  let one   = Num 1
      two   = Num 2
      three = Num 3
      mul   = Mul two three
      add   = Add one mul
    in Rose   (Bounds (0,0) (9,9), add)
        [Rose (Bounds (0,0) (1,2), one)  []
        ,Rose (Bounds (3,4) (9,9), mul)
        [Rose (Bounds (3,4) (5,6), two)   []
        ,Rose (Bounds (7,8) (9,9), three) []
        ]
      ]
```

This approach has some advantages:

- The datatype is simple and easy to understand.
- Although the above expression is pretty big for such a simple example, construction of such trees during parsing can be made easy by hiding the details in a monad transformer, exposing a function to introduce branches.
- Mapping from tree selection to text selection is trivial: simply select the *Bounds* value of the rose node.
- Mapping from text selection to tree selection can also be done: walk the tree, finding the node whose bounds correspond to the query range. The searching can be implemented efficiently, pruning complete subtrees if their bounds do not surround the search range.
- *Rose* is a regular datatype and a corresponding zipper type *RoseZipper* to express structural selections is easily created.

However, this approach also has some serious disadvantages:

- Subexpressions are repeated in their entirety in subtrees. While this is not a memory problem—there is plenty of sharing going on—storing trees in this way causes a lot of redundancy and potential for invalid parse trees. The type offers no guarantees that subtrees are actual subexpressions. It also complicates updating such trees, as they have to be modified in various places in order to be kept consistent.
- The consumers of expressions that want to use the position information have to work on parse trees instead of values of type *BareExpr*. This makes these algorithms significantly less elegant. Similarly, functions working on zipper values have to work on *RoseZipper*s instead of *BareExprZipper*s.

To illustrate these problems, take a look at the following evaluation function for expressions. We want the function to work on parsed expressions and to return an error on division by zero, indicating where in the input sentence the error occurred. Therefore the function takes a *ParseTree BareExpr* as input rather than a vanilla *BareExpr* so that position information is available, and it returns either a *Bounds* or an *Int*.

```
eval :: ParseTree BareExpr → Either Bounds Int
eval (Rose (bounds, expr) cs) =
  case (expr, cs) of
    (Num n,   [])     → pure n
    (Add  _ _, [x,y]) → (+) <$> eval x ⊛ eval y
    (Sub  _ _, [x,y]) → (−) <$> eval x ⊛ eval y
    (Mul  _ _, [x,y]) → (∗)  <$> eval x ⊛ eval y
    (Div  _ _, [x,y]) → do
```

```
$x'$    $\leftarrow$ *eval x*
$y'$    $\leftarrow$ *eval y*
**case** $y'$ **of**
  0 $\rightarrow$ *Left bounds*
  _ $\rightarrow$ *pure* ($x'$ '*div*' $y'$)
```

The problems are immediately apparent. Firstly, evaluation has to be written in applicative or even monadic style to propagate potential errors.[1] Secondly, the fields of the *BareExpr*s are simply discarded. Instead the list of child parse trees is pattern-matched against a fixed number of expected children, depending on the specific constructor. This indicates that the fields are simply not necessary; then why were they set in the first place? Furthermore, cases where the number of child parse trees fails to match expectations result in a runtime pattern match error. Functional programmers like it when the type system ensures safety, but the datatype we have here contains redundant information and allows bad shapes.

We can do better than this.

[1]Note that a non-standard **instance** *Monad* (*Either e*) is used: one with no constraint on the error parameter *e*.

Chapter 5

Annotated base functors

The approach above was not satisfactory because we left the definition of *BareExpr* untouched; we only used it. Let us see what happens if we allow ourselves to change *BareExpr*'s definition.

Our goal is still to add position information to every node. We can do this in two ways: either we add a new field to every constructor, or we couple each *use* of *BareExpr* with *Bounds*. We choose the latter because it allows the extraction of a nice abstraction.

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

Notice that *BareExpr* and *PositionalExpr'* look very much alike: they have the same structure. Only the types of their recursive positions are different. We can maximize 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 ExprF r
    = Num Int
    | Add  r r
    | Sub  r r
    | Mul  r r
    | Div  r r
```

We can now redefine *BareExpr* in terms of *ExprF*. Remember that the type argument of *ExprF* determines the shape of its children. To reconstruct the original expression type, we want the children themselves to be expressions as well.

Therefore need to give the expression type we are defining itself as type argument to *ExprF*. This leads to a recursive definition of *Expr*. We can encode such a definition by a new datatype:

```
newtype Expr = Expr (ExprF Expr)
```

Expanding *Expr* repeatedly leads to the infinite type *ExprF* (*ExprF* (*ExprF*...)), indicating that the obtained tree is of *ExprF*-shape at every level.

PositionalExpr can be expressed in terms of *ExprF* in a similar way. To insert the position information at every level, we wish to obtain the infinite type (*Bounds*, *ExprF* (*Bounds*, *ExprF*...)). We need another datatype to achieve this.

```
newtype PositionalExpr = PositionalExpr (Bounds, (ExprF PositionalExpr))
```

The idea of abstracting over a type's children like this is not new and goes by several names, including *open recursion*. The *ExprF* version of the expression datatype is often called the *base functor* and we will use that name in the rest of this thesis. The suffix *F* to indicate the functor version of a datatype comes from *Polytypic Programming* (JJ96).

The fact that types such as *Expr* and *PositionalExpr* use themselves as arguments to functors in their own definitions is often made explicit by expressing them in terms of the well-known datatype *Fix*. Doing so adds to the modularity and enables some generic functions, which we will take a look at in a moment. An early document discussing the *Fix* datatype is *Recursive types for free!* (Wad90).

The *Expr* datatype is immediately expressible in terms of *Fix*; we redefine it here:

```
newtype Fix f = In   {out     :: f (Fix f)  }
newtype Expr = Expr {runExpr :: Fix ExprF}
```

Because number literals and arithmetic operators are overloaded in Haskell, we can easily construct values of the new *Expr* type if we supply an appropriate **instance** *Num Expr* and **instance** *Fractional Expr*:

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

To redefine *PositionalExpr* in terms of *ExprF* we need to go through a bit more trouble. We introduce a new datatype we can use for adding the position information at every level before we give it to *Fix*, somewhat like a tuple type that is lifted on one side:

```
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 PositionalExpr =
  PositionalExpr {runPositionalExpr :: Fix (Ann Bounds ExprF)}
```

These last definitions of *Expr* and *PositionalExpr* are the final ones for this section and they will be used throughout the following sections. Furthermore, we introduce two type synonyms that will prove useful later on.

$$\begin{aligned}&\textbf{type}\ \mathit{AnnFix}\ \ x\ f = \mathit{Fix}\ (\mathit{Ann}\ x\ f)\\&\textbf{type}\ \mathit{AnnFix}_1\ x\ f = f\ (\mathit{AnnFix}\ x\ f)\end{aligned}$$

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:

$$\begin{aligned}&\mathit{mkAnnFix} :: x \rightarrow \mathit{AnnFix}_1\ x\ f \rightarrow \mathit{AnnFix}\ x\ f\\&\mathit{mkAnnFix}\ x = \mathit{In} \circ \mathit{Ann}\ x\end{aligned}$$

There are numerous advantages to expressing *Expr* and *PositionalExpr* this way, some of which we have already seen. The most important one is that we have solved one of the issues with the approach in the previous section: we no longer use unbounded lists for the children, so we cannot build syntax trees anymore that do not actually correspond to expression trees and have too many or too few children. We are also not storing complete subtrees in separate fields anymore and so do not have any redundant information in our datatype. The expression parts and annotation parts are tightly interwoven, ensuring that all values of type *AnnFix Bounds ExprF* are valid.

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

$$\begin{aligned}&\mathit{unannotate} :: \mathit{Functor}\ f \Rightarrow \mathit{AnnFix}\ x\ f \rightarrow \mathit{Fix}\ f\\&\mathit{unannotate}\ (\mathit{In}\ (\mathit{Ann}\ \_\ \mathit{tree})) = \mathit{In}\ (\mathit{fmap}\ \mathit{unannotate}\ \mathit{tree})\end{aligned}$$

To use this function on annotated expressions, we need *ExprF* to be an instance of *Functor*. We omit that instance here because it is trivial: there is only one implementation that adheres to the functor laws.

5.1 Catamorphisms over fixed points

Now that we have added the fields for storing position information to our expression datatype it is time to adapt the producers and consumers of the new expression types. We start with the consumers, specifically the *catamorphisms*.

The application of fixed points in catamorphisms is well-known, but because it is so important for the rest of the story, we describe it again in this section. A more formal treatment is offered in *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* (MFP91).

To understand what catamorphisms are, take a look at the use of the best-known one: *foldr* over lists.

```
foldr :: (a → b → b) → b → [a] → b
```

One way to understand the behavior of this function is to notice it replaces the two list constructors (:) and [ ] 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* (⊕) *e* turns the list $x : y : z : [\,]$ into $x \oplus y \oplus z \oplus e$.

The same can be done for any other algebraic datatype. For the *Expr* datatype, for example, we can create a function *cataExpr* that takes five arguments: one for every constructor. The function then recursively traverses any expression we give it, applying the appropriate functions to the fields of the constructors.

Instead of giving the five arguments separately to the function, we can 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 then becomes:

```
cataExpr :: ExprAlg a → Fix ExprF → 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 *BareExpr*'s definition: every constructor of *BareExpr* has a corresponding constructor in *ExprAlg*, and each constructor in *ExprAlg* has fields that directly correspond to those of *BareExpr*'s constructor. Can we do the same trick as before and find a suitable abstraction? It turns out we have yet another use for *ExprF*: the types *ExprAlg a* and *ExprF a* → *a* are isomorphic. We can show this using some 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 powers b^a.

$$ExprF(a) = Int + a^2 + a^2 + a^2 + a^2$$

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

We see that *ExprAlg a* and *ExprF a* → *a* are isomorphic. Let us see how using this new type changes the definition of *cataExpr*:

```
cataExpr :: (ExprF a → a) → Fix ExprF → 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 *ExprF*'s constructors: this part is hidden in the function argument and the call to *fmap*. In fact, there is nothing anymore in this definition that is specific to *ExprF*. Therefore, *cataExpr*'s signature as it is above is too specific, both in name and in type. Writing the body in point-free 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
```

This means that for every datatype that is defined in explicit-recursion form, custom datatypes for their corresponding algebras and custom functions for their catamorphisms are no longer necessary: yet another benefit of reusing existing building blocks.

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

```
exprEval :: Algebra ExprF 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

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

Because *cata* takes care of the recursive positions, the algebra can assume the fields already contain the results of the evaluation.

5.2 Error algebras

The catamorphisms above did not take the possibility of failure into account. Neither did they do anything with position information. Let us fix both of those issues. We have our evaluation algebra return an *Either String Int* to indicate failure or success. Furthermore, instead of working on *ExprF*, we have the algebra accept *Ann Bounds ExprF* (which is a *Functor* because *ExprF* is a *Functor*) so that we have position information available in case things go wrong. The algebra then looks like this:

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

Although the problem of redundancy and invalid shapes is gone, this algebra still suffers from the other problem the catamorphism at the end of section 4 had: computations have to be written in applicative or monadic style. 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*) and so assumes that when the catamorphisms were applied to children, *they were successful* and produced *a*’s instead of error values. This means that it is no longer necessary to use applicative style in the algebras and evaluation is once again pretty:

```
exprEval :: ErrorAlgebra ExprF 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 in the tree, popping up as the result at the root.

There are situations where several children simultaneously produce errors. Rather than arbitrarily picking one of the errors to bubble up, we *mappend* them together, introducing a *Monoid* constraint on the error type *e*.

Of course we cannot give error algebras to our generic *cata* function just like that, because *cata* expects normal algebras. Also, the applicative computations have not simply gone; they just need to be applied outside of the algebra.

In *Applicative programming with effects* (MP08), McBride and Paterson show how to generically capture applicative computations over functors using the type class *Traversable*. That type class essentially provides one function:

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

Traversing *ExprF*, for example, looks like this:

```
instance Traversable ExprF 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 *ExprF*'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 give to *cata* again.

```
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 Paterson and McBride. It is similar to *Either*, but it is designed to only be used in an applicative way so that sequencing two errors results in the sum 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 a moment ago. Here is the datatype and its *Applicative* implementation:

```
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 pretty algebras with error functionality, we have not addressed yet what to do with annotations a tree might have. To do something useful with the annotations, we need a new catamorphism function; one that works on *AnnFix*'s instead of normal *Fix*'s. If we have this new function take error algebras too, we can automatically couple potential errors with the annotations at the positions at which those errors arose, regaining all the functionality that the inelegant catamorphism above had:

```
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
```

5.3 Parsing annotated values

Now that we have adapted the consumers of the new, annotated datatypes, it is time to adapt the *producers* of the annotated datatypes to automatically fill in the annotations. There are many kinds of producers. One example is the palette in a graphical editor that creates new model objects. But the most popular producer by far is the parser that converts plain text to model objects, introducing structure.

For that reason we focus only on adapting parsers. Haskell is well-known for the variety of parser libraries and tools that are available for it. We will choose one of these and show how to easily create values of the new expression type.

Because we are developing solutions to be used in pure Haskell, we limit our options to parser *libraries* only, excluding preprocessing tools such as Alex and Happy so that we can manipulate the parsers as first-class citizens in Haskell. Even then there are several options, including polish parsers developed at Utrecht University, Parsec and the ReadP library that ships with the standard libraries.

It does not really matter which one of these we pick. Although there are some differences between the libraries, they are all very good at that part we are interested in: the actual parsing. For this thesis we have chosen to work with Parsec as it probably the best-known of those options. We will use version `parsec-3.0.1`, available from Haskell's package repository Hackage[1]. From

[1] `http://hackage.haskell.org/package/parsec-3.0.1`

here on we assume the reader is familiar with the most important Parsec combinators.

5.3.1 A parser for *BareExpr*

In order to properly compare the parser for the new version of our expression datatype with that of the old version, we need to give both versions. We have given neither so far. Let us start with the old version for *BareExpr*, the version without the use of fixed points.

One of the fundamental parsing operations is the parsing of a single symbol. There are various strategies for this used by different libraries. The parsers from UU, for example, allow you to supply a range of symbols to accept, specifying the lower and upper bound and using the *Ord* constraint to test membership of the range. Sometimes symbol equality is used, forcing the user to combine many smaller parsers if a range of symbols is to be accepted.

Parsec uses the most flexible of all the possible approaches: a predicate of type *symbol* → *Bool* that tells whether a symbol is to be accepted. The function that accepts this predicate and produces the appropriate parser is often called *satisfy*; an early example of this can be found in (Hut92). This approach is the most flexible because two other approaches from the previous paragraph can be expressed in terms of *satisfy*. [2]

Our expression producer consists of two phases: the lexer converting characters to expression tokens and the actual parser converting these tokens to expression trees. Using two phases like this makes the second phase easier because we can discard all the whitespace and comment tokens from the input so that the parser does not have to bother with them. Our token type is as follows:

```
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 not very interesting and unimportant for the rest of the story, so we omit it. That leaves only the parser:

[2]While most flexible, *satisfy* also gives the parsing library the least amount of information and precludes certain optimizations that otherwise would have been possible.

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

5.3.2 Keeping track of the position during parsing

Now that we have a parser for the old *Expr*, we can build one for the new *Expr* and compare them.

The new parser will have to use the constructors from *ExprF* (which have the same names as those of the old *Expr*) and insert position information at every level before the trees can be used as children of constructors higher up in the tree.

To properly annotate the values we build during parsing, 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 us to keep track of ranges of whitespace around token sequences. It therefore makes sense to use a range of whitespace as our position information at any moment during the parsing, so we set *Range* as the type of the state parameter *u* in *ParsecT s u m a*, updating the range every time a token is parsed.[3]

The easiest way to have the position information available every time we read a token is to couple each token in the input stream with its *Bounds*. Computing the proper bounds for each token needs to be done before discarding the whitespace tokens from the lexer's output, because we need the whitespace tokens to compute the margins. Therefore we combine the discarding of these tokens and the computation of the bounds in a single operation called *collapse*:

```
collapse :: Symbol s ⇒ (s → Bool) → [s] → [(s, Bounds)]
collapse space ts = collapse′ (0, symbolSize lefts) space rest
  where
    (lefts, rest) = span space ts
collapse′ :: Symbol s ⇒ Range → (s → Bool) → [s] → [(s, Bounds)]
collapse′ _ _ [] = []
collapse′ left space (t : ts) = new : collapse′ right space rest
  where
    (_, leftInner) = left
    rightInner     = leftInner  + symbolSize t
    rightOuter     = rightInner + symbolSize rights
    right          = (rightInner, rightOuter)
    (rights, rest) = span space ts
    new            = (t, Bounds left right)
```

Most of the work is done in *collapse′*. Its first argument is its current offset in the stream, the left margin of the bounds of the next token. The right margins,

[3]Type constructor *ParsecT*'s arguments *s*, *u*, *m* and *a* stand for stream type, user state, underlying monad and result type, respectively.

which form the left margins in the recursive call, are computed by asking for symbol sizes: the reason for the *Symbol* type class, which we will look at in more detail in a few moments. The function *collapse* is the one that is exposed to clients of the API and does not need an offset parameter because it assumes it is at the start of the input. Apart from the input stream, the only other argument is a function that tells which of the symbols are to be discarded. By expressing this as a function $s \to Bool$, *collapse* assumes this can be decided locally by looking at the symbol in question alone. If a context-sensitive decision is needed, *collapse* could be adapted to a more complicated scheme.

Rather than building *collapse* specifically for *ExprToken*s, we look at what properties we need of token types and capture these in a type class called *Symbol*:

```
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 the one we used above. Its default implementation may be overwritten for efficiency, if necessary.

The observant reader may have noticed that *symbolSize* is called on both single symbols and lists of symbols in the definition of *collapse*. This is possible because an extra instance is provided for lists:

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

To recap, the new parser will have *Range* values as user state and will consume tokens coupled with their position information. This is reflected by a new type synonym *P*:

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

The type constructor *ParsecT* has been given only two arguments and still expects its third and fourth argument *m* and *a*, so to fully specify *P*, it needs three arguments *s*, *m* and *a*.

Every time a new token is consumed, the state needs to be updated. We can hide this in the new definition of *satisfy*. That function will be defined in terms of one of the fundamental building blocks of Parsec parsers: the function *tokenPrim*. Its type is:

```
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 not really use this information during parsing, but it is still used by Parsec when generating error messages. For that reason, we will update it as best as we can using the position information we maintain in the state. The third argument is the predicate passed to *satisfy* that tells which token is expected. Our implementation of *satisfy* then becomes:

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

Figuring out the current position in the stream is now a simple matter of asking for the current user state:

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

5.3.3 Building recursively annotated values

Now that we have the parsing infrastructure in place, it is time we look at how to build *AnnFix* values. Let us look at the simplest expression possible and the only ones that form leaves in expression trees: number literals. In *ExprF*, the constructor for number literals has type:

```
Num :: Int → ExprF r
```

A number with its number field set, such as *Num* 1, has a type that can be specialized to $AnnFix_1$ *Bounds ExprF*. Now we just need to wrap the *Bounds* around it using *mkAnnFix*. We ask for the left margin right before parsing the literal token and the right margin after:

```
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)
```

Instead of putting this in one function, we have fleshed out some operations we can reuse in a moment. Firstly we introduce a new type synonym *ExprParser* for our expression parser: it will consume *ExprToken*s and use an underlying *Identity* monad. The implementation of *pNum* is equal to the one in the old parser except for the call to *unit*. Function *unit* takes a parser that yields an $AnnFix_1$ and turns it into a parser that yields an *AnnFix* by wrapping the position information around it. This is a useful combinator for parsers that produce simple nodes such as number literals.

Even if a parser does not produce simple nodes, it is very often the case that the call to retrieve the right margins and the call to *mkAnnFix* are right next to each other. This is reflected in *mkBounded* which, when given the left margin, asks for the right margin itself and then builds an *AnnFix*.

In the old parser, the branches of the expression trees were built using *chainl1*. Since we were not annotating values back then, there was no distinction in type between values with and without annotation like there is now. We can adapt *chainl1* to make that distinction explicit.

```
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)
```

Here the first argument is again the parser for the operands and the second the parser for the binary operator. In our examples a typical operator is $Add :: r \rightarrow r \rightarrow ExprF\ r$. If we give annotated children to *Add*, we end up with $AnnFix_1$'s again, which is reflected in *chainl1*'s type. It is *chainl1*'s responsibility to insert the right position information. The full definition of *chainl1* follows:

```
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
```

Apart from the new *chainl1*, nothing else in the old parser needs to change. In fact, 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
```

Of course, this is only a small example and there are numerous other combinators commonly used in parsing. However, all of these can be adapted to position-saving variants as the API of building annotated values is very general. Just for good measure, here is the position-saving version of *chainr1*:

```
chainr1 :: 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)
chainr1 px pf = fix $ λloop → do
  left ← getPos
  x    ← px
  option x $ do
    f ← pf
    y ← loop
    mkBounded left (f x y)
```

Note the differences between *chainr1* and *chainl1*: in both cases the parsing is done in a right-recursive way because Parsec does not like left-recursive grammars. In the case of *chainl1*, however, the *value* is built in a left recursive way by making the left-hand side an argument to the loop (called *x* in the code). Another important difference is that in *chainl1* the call to *getLeftBounds* is outside the loop while in *chainr1* it is inside.

5.4 Exploring annotated trees

5.4.1 Representing structural tree selections

Given an annotated subtree of type *AnnFix x f*, we can easily find the corresponding text selection: simply extract the *Bounds* value in the *Ann* constructor. To do the conversion in the other direction, we need to search the tree for a node whose bounds match the text selection. In this section we will introduce functions that do just that.

But what should be the result of such a function? It could just yield the subtree that was selected, but then the context of that subtree is lost. Grote Trap solved this by returning the path from the root to the subtree, modeled as a list of child indices [*Int*]. This gives enough context, but it is untyped: such a path could apply to any tree, and you cannot be sure the path is actually valid for a certain tree until you follow it down to the selected node.

The traditional, functional way for representing structural selections is the zipper structure, described by Gérard Huet in 1997 (Hue97). A zipper datatype is derived from another datatype: what the zipper looks like exactly depends on the shape of the original datatype. For example, the zipper for lists is different from the zipper for the arithmetic expressions we have been using as examples. Conor McBride showed how to automatically find out what a datatype's zipper type looks like (McB01).

A value of a zipper type represents the selection of one particular node in a tree. The selected node is called the zipper's *focus*. Once you have such a zipper value, you can move around 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. A zipper value is a primary data source: you don't need to hang on to the original, selection-less tree because it is implicit in the zipper value.

Our programs so far have been generic over the particular shape functor f, albeit under some class constraints to have access to certain functions. Can we also generically derive zippers from functors using this technique? We are unsure whether this is possible and what the type class constraint would look like.

However, we can approximate the idea of the zipper and build one generically for functors if we give up the possibility of $O(1)$-time updates of the zipper's focus. The data structure looks like this:

```
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 terms of base functors. However, in this section we will only construct and use zipper values of type *Zipper* (*Fix f*), for some functor f.

The reason that *Zipper* does not allow $O(1)$ updates is because its values contain cycles and Haskell does not allow destructive updates. If we were to change the value of a zipper's *zFocus* field, it would no longer be a consistent value, because the other fields—*zUp*, *zLeft*, *zRight* and *zDown*—would still point to old zipper values. To make the zipper consistent, all these values would have to be rebuilt, recursively, resulting in a completely new zipper. This takes longer than $O(1)$ time. The original, true zipper does not have this problem because it contains no redundant information and no cycles.

Despite this deficiency, *Zipper* is still usable as a representation of structural selections: there is the focus on a particular node, and there is also context available. Let us look at how to build a zipper value from a *Fix f*. We start by giving the type signature:

```
enter :: Foldable f ⇒ Fix f → Zipper (Fix f)
```

There is a new class constraint on the functor type f called *Foldable*, available in the standard libraries in **module** *Data.Foldable*. In power, *Foldable* lies between *Functor* and *Traversable*: every *Traversable* is *Foldable*, and every *Foldable* is a *Functor*, as reflected by the type classes' super classes. The heart of *Foldable* is the *fold* function:

```
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 $toList :: Foldable\ t \Rightarrow t\ a \rightarrow [a]$ does.

For our fixed point functors, *f*'s elements are its children, and therefore we can use *toList* to obtain the functor's children, which is exactly what we need when constructing zippers.

The *enter* function is defined in terms of a helper function:

```
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 one) of the node that will be produced, and its second argument is its left sibling (if it exists). The third and final argument is the list of the resulting node's right siblings, still to be processed. The **where**-clause builds the current focus and the recursive values and gives them names, to be passed on to the recursive function calls. Building the tree in this way ensures optimal sharing, a process that is sometimes called *tying the knot*.[4]

That sharing is optimal can be demonstrated using *Vacuum*, a Haskell library for visualizing the heap. Figure 5.1 shows the zipper for the parse tree of `2 + 3 * 4` using the UbiGraph frontend for Vacuum.[5] The fact that the tree is rendered is proof that the zipper structure—even though it is cyclic—consumes finite space, because otherwise Vacuum would not produce any results.

Once the zipper structure has been created, it can be traversed using the record selectors *zDown*, *zUp*, *zLeft* and *zRight*. From anywhere in the zipper, we can recover the original tree again 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*:

[4]More examples of *tying the knot* can be found at `http://www.haskell.org/haskellwiki/Tying_the_Knot`.

[5]`http://hackage.haskell.org/package/vacuum-ubigraph`

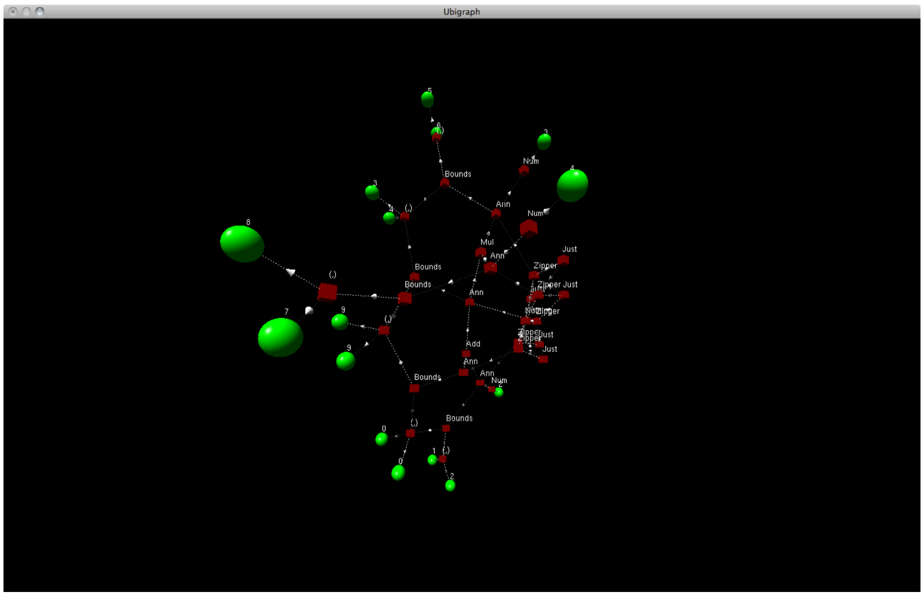


Figure 5.1: A visualization of the zipper for the parse tree of `2 + 3 * 4` using the `vacuum-ubigraph-0.1.0.3` package.

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

The zippers make the traversal functions that are discussed in the following sections easier to define and understand.

5.4.2 Annotation-guided exploring

Now that we know the return type of the function that converts text selections to tree selections, we can actually build such a function. So far we have built functions that abstract over the specific annotation type, rather than specialized functions for trees annotated with position information. We will do that again here.

A naive implementation of the conversion would visit the entire tree, starting at the root and at each node searching recursively down and to the right until a node is found whose bounds match the query range. But the laws outlined in section 3 give us a lot of information and allow us to prune entire subtrees in some cases:

- If the left offset of the query range is strictly less than the current node's inner right offset, we know we do not have to look at the node's right siblings because law 2 says 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 know 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 type $x \rightarrow \mathit{ExploreHints}$, 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. If the query range matches these bounds, which of the two nodes should then be chosen? We will go for that node that is the deepest. But we cannot make this decision in general: if we abstract over the annotation type, we cannot make any assumptions about the domain anymore. For that reason, the general function will return the full list of matching tree selections.

Our complete exploration function looks like this:

```
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 bounds@(Bounds _ (ir, _)) =
    ExploreHints
      { matchHere   = range ‘rangeInBounds‘ bounds
      , exploreDown = range ‘rangeInRange‘ outerRange bounds
, exploreRight  = left ⩾ ir
}
```

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

Another use case is selecting a single position within the text, rather than a range. To that end we will also define a *selectPos*, but as before we will first define a generalized version and then define *selectPos* in terms of it.

```
findLeftmostDeepest :: Foldable f ⇒
  (x → Bool) → (AnnFix x f) → Maybe (Zipper (AnnFix x f))
findLeftmostDeepest down = listToMaybe ∘ reverse ∘ explore hints
  where
    hints x
      | down x    = ExploreHints True  True  False
      | otherwise = ExploreHints False False True
```

Rather than using *ExploreHints* again, a *Bool* suffices in this case: given an annotation *x*, should we go down or continue searching to the right? Such a query is easy to express in terms of *explore*. Again, *explore*'s result is reversed before wrapping the head of the list in a *Just*. Now *selectPos* can be written as follows:

```
selectByPos :: Foldable f ⇒ Int → AnnFix Bounds f → Maybe (Zipper (AnnFix Bounds f))
selectByPos pos = findLeftmostDeepest (posInRange pos ∘ innerRange)
```

5.4.3 Repairing and navigating text selections

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

Section 1.1 distinguished 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. But what if a text selection is invalid? Invalid selections are not completely 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. That is exactly what *repairBy* does:

```
repairBy :: (Foldable f, Ord dist) ⇒
  (Range → Range → dist) → 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. Then it asks for all the selections that would have been valid using *validBounds* and sorts them according to some cost function, to which inner bounds are given. For this *sortOn* is used, which sorts a list based on a property of all elements in the list.[6] Then 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 with 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 would like to express navigation based on text selections. Suppose the user has selected a piece of text that corresponds neatly to a structural selection. Now the user wants the selection to expand to the direct parent of the selected node.

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 in a function if we can express the act of moving around within a zipper in a type.

This type is not hard to find: it is exactly the type of the record selectors of *Zipper*. All four selectors have the type *Zipper a* → *Maybe* (*Zipper a*). We can compose such functions with the Kleisli arrow composition operator (>=>) :: *Monad m* ⇒ (*a* → *m b*) → (*b* → *m c*) → (*a* → *m c*). 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)}
```

Besides composition of *Nav*s, we can also think of an identity navigation: staying in the same position. This makes *Nav* a nice *Monoid*:

```
instance Monoid Nav where
  mempty                      = Nav return
  mappend (Nav n1) (Nav n2) = Nav (n1 >=> n2)
```

Now that the type of navigations is known, we can write navigation based on text selections as follows:

[6]Luke Palmer has written a blog post on *sortOn*: `http://lukepalmer.wordpress.com/2009/07/01/on-the-by-functions/`

$$
\begin{array}{l}
\mathit{moveSelection} :: \mathit{Foldable}\ f \Rightarrow \mathit{AnnFix}\ \mathit{Bounds}\ f \rightarrow \mathit{Nav} \rightarrow \mathit{Range} \rightarrow \mathit{Maybe}\ \mathit{Bounds} \\
\mathit{moveSelection}\ \mathit{tree}\ (\mathit{Nav}\ \mathit{nav})\ \mathit{range} = \\
\quad (\mathit{rootAnn} \circ \mathit{zFocus}) <\$> (\mathit{selectByRange}\ \mathit{range}\ \mathit{tree} \gg= \mathit{nav}) \\
\mathit{rootAnn} :: \mathit{AnnFix}\ x\ f \rightarrow x \\
\mathit{rootAnn}\ (\mathit{In}\ (\mathit{Ann}\ x\ \_)) = x
\end{array}
$$

Naively, *moveSelection* would return *Maybe Range* rather than *Maybe Bounds*, but *Bounds* contains strictly more information, and we return all the information we have after finding the newly selected node.

5.5 Summing up

One of the disadvantages of the approach taken in this section is that we had to adapt our original *Expr* datatype to make the type of its children variable, while our original desire was to develop a solution for adding position information without having to change anything about the existing datatype.

However, in return for this sacrifice we have gained many benefits. By building the datatypes we require from smaller building blocks (*ExprF*, *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 *Expr* did not need many changes to work with the new annotated expressions because most of the work was hidden in the combinators. The building blocks also allowed us to generically express structural selections (zippers) and exploration functions. 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.

There is one major disadvantage that was introduced by switching to the explicit recursion method: we can no longer work with families of datatypes. We have not noticed this before because our examples so far have only focused on arithmetic expressions, which do not need mutually recursive datatypes. In *Data types à la carte* (Swi08), Wouter Swierstra 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: freedom we do not want, because we want to specify which exact datatype to recurse into. In the next section we will look at generic programming techniques to solve this problem properly.

Chapter 6

Annotations in MultiRec

In this section we will reimplement the annotation framework of the previous section in the generic programming library *MultiRec* (RHLJ09). In a sense, MultiRec is more generic than the open recursion approach we took in the previous section, offering more flexibility but also more complexity. We will see why in a moment.

Before we look at MultiRec in particular, let us have a short introduction to generic programming in general: what is it and why would you implement the same functionality in multiple generic libraries?

6.1 Introduction to generic programming

The report *Comparing Libraries for Generic Programming in Haskell* (RJJ$^+$08) gives a very nice explanation of the term *generic programming*:

> Software development often consists of designing a datatype to which functionality is added. Some functionality is datatype specific. Other functionality is defined on almost all datatypes, and only depends on the structure of the datatype; this is called datatype-generic functionality. Examples of such functionality are comparing two values for equality, searching a value of a datatype for occurrences of a particular string or other value, editing a value, pretty-printing a value, etc.

In Haskell 98, datatype genericity is available for a limited number of functions as **deriving** clauses following datatypes.[1] The clause **deriving** *Eq* following the definition of **data** *BareExpr*, for example, makes available an operator $(\equiv) :: BareExpr \to BareExpr \to Bool$ for comparing two values of type *BareExpr*.

To implement such 'deriving' behavior for a custom type class, one would have to extend the compiler and analyze the datatype in question. Because this is often a lot of tedious work, programmers have looked for suitable abstractions

[1]See the Haskell 98 Report, section 10 "Specification of Derived Instances".

in which programming this generic behavior over datatypes is made easier, by providing different interfaces on this low-level view on datatypes: thus generic programming was born. Because these interfaces are often less flexible, they can be made substantially simpler and easier to use.

Of all these interfaces, Template Haskell (SP02) is one of the most powerful because it aims to offer a one-to-one mapping between the syntax of a Haskell datatype and runtime values describing such a datatype. It allows for generic programming because—among many other things—it lets the programmer write a function that takes a datatype as input and produces some function declarations. This function is evaluated at compile-time, and the produced declarations are then compiled into the final program. This is efficient: there is some overhead at compile-time, but the generated code is as fast as if it had been written by hand.

Template Haskell has a few disadvantages, however. Because it is so powerful, it is also reasonably verbose and even simple generic functions easily span many lines. In fact, there is so much freedom that it is possible to build type-incorrect or otherwise nonsensical expressions and declarations. Errors in the resulting expressions are caught and reported at compile-time, but because they concern generated code, it is very difficult to give accurate position information about where those errors occurred.[2] That the errors often concern generated names complicates matters even further.

So when code generated by Template Haskell is compiled, the compiler will not fail during the parsing phase, but it might during the type-checking phase. This did not sit well with some programmers: Haskell is known for its powerful type system; was it not possible to create an approach to generic programming directly in Haskell, as a library, so that it is necessarily type-safe? And so the search for good generic programming strategies continued.

Some of the resulting generic libraries are Scrap Your Boilerplate (SYB) (LP05), Uniplate (MR07), Extensible and Modular Generics for the Masses (EMGM) (OHL06) and the various *compos* operators (BR06). These approaches express their generic functionality in terms of type classes; generic programs then depend on the existence of type class instances of the types they are expected to work on. The programmer can write instances for these types by hand, but most of these libraries offer a way to bootstrap the generic functionality. SYB, for example, allows instances to be created by a **deriving** clause through a compiler extension. Uniplate, in turn, has an implementation in terms of SYB.

So which of these libraries should you choose for your own generic program? The differences between these libraries lie mainly in how they deal with families of datatypes, fields of different types, lists (or other collections) of children, type arguments (if at all) and information about constructors such as their name and—if they are operators—their fixity.

None of these libraries, however, allow the user to generically change the *shape* of a datatype. Their functionality is restricted to *queries* that compute some value over a tree (like a fold) or *transformations* that allow rewriting tree terms.

[2]Something we obviously value very much in this thesis. :-)

In section 5 we saw how we can write a datatype in such a way that we can later decide what type the children will have by introducing an extra type argument. This added flexibility turned out to be crucial in adapting a datatype to hold position information.

Although the degree in which *ExprF* exposes its structure is very useful, it is still very limited, from a generic programmer's point of view. For example, we cannot generically count the number of constructors of a datatype, or count the number of fields in a specific constructor, or generate members of a datatype, or check whether two types are equal. In the context of type-indexed datatypes there is also more structure to be exposed. Zippers, for example, cannot be expressed in terms of a base functor.

To divulge more of a type's structure, we can model it using a fixed set of building blocks. All datatypes follow the same pattern: they have a number of constructors and each constructor has a number of fields. This way of modeling datatypes is called *sums of products*, because of the way the number of values of a datatype add up: a single constructor has zero or more fields. Each of these fields individually has a number of values it can take on. The constructor can be initialized with any combination of these values, so the number of possible instantiations equals the product of each field's number of values.

A datatype consists of several constructors. To compute the number of values in a datatype, we take the numbers of values of each constructor and add these up. In this sense, a datatype is a sum (the constructors) of products (the fields of the constructors).

Using sums of products is very interesting for our application because it too, just like the approach in the previous section, allows the use of fixed points and thus the insertion of annotations at every level.

ASTs often comprise several datatypes where one datatype refers to another. Sometimes they are even mutually recursive. Until recently it was unknown how to model such a *family* of datatypes using sums and products without knowing in advance how many datatypes the family would consist of. MultiRec, however, offers a solution to this problem in a type-safe way. Here by 'type-safe' we mean that it is impossible for the sums-of-products model of the datatypes to create trees which cannot be written using the original datatypes (i.e. they are completely isomorphic, up to $\perp$) and that nowhere in the implementation any 'unsafe' functions (such as *unsafeCoerce*) are used.

This is why MultiRec is an interesting generic programming library for our problem: it allows for families of possibly mutually recursive datatypes, exposes a lot of information about the datatypes using sums of products, and uses fixpoints to boot!

Let us take a look at some concrete examples.

6.2 Pattern functors

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
  = EAdd    Expr Expr
  | EMul    Expr Expr
  | ETup    Expr Expr
  | EIntLit Int
  | ETyped Expr Type
  deriving (Eq, Show)
data Type
  = TyInt
  | TyTup Type Type
  deriving (Eq, Show)
```

We have gotten rid of the constructors for division and subtraction and in their place have two other constructors: one for creating tuples of expressions and one for expressions with type signatures. The latter uses a separate type *Type* to describe expression types. The tuple constructors of both *Expr* and *Type* allow arbitrary nesting, e.g. $(1,(2,3))$ and $(Int,(Int,Int))$ can be modeled as *ETup* (*EIntLit* 1) (*ETup* (*EIntLit* 2) (*EIntLit* 3)) and *TyTup TyInt* (*TyTup TyInt TyInt*), respectively. The type annotations introduce some ambiguity to the language so that we have reason to type-check expressions and possibly report errors later when we look at MultiRec's version of error catamorphisms.

Let us translate these datatypes to sums of products. We will use `multirec-0.4`, available from Hackage[3]:

```
type PF_Expr =
       I  Expr :*: I Expr :*: U
  :+: I  Expr :*: I Expr :*: U
  :+: I  Expr :*: I Expr :*: U
  :+: K Int    :*: U
  :+: I  Expr :*: I Type :*: U
type PF_Type =
       U
  :+: I  Type :*: I Type :*: U
type PF_Tuples = PF_Expr :>: Expr :+: PF_Type :>: Type
```

Here the *Tuples* in PF_{Tuples} is the name we have given to our family of datatypes, comprising of *Expr* and *Type*. The sums-of-products version of a family of datatypes is called the family's *pattern functor*; that is what *PF* stands for. The pattern functors of either type are written separately as PF_{Expr} and PF_{Type}. Their structure mirrors the structure of the corresponding datatypes directly. The

[3]`http://hackage.haskell.org/package/multirec-0.4`

two pattern functors are then tagged and combined in PF_{Tuples}. In all, this type is pretty large but consists only of six distinct building blocks:

- :+: is used to denote choice (sums), be it between datatypes of the family or constructors of a particular datatype.
- :∗: can be seen as the cons in the list of constructors, forming the products. Just like the cons operator : on lists, this operator associates to the right.
- *I xi* is used to indicate recursion into type *xi* of the family in question. It always appears on the left-hand side of a :∗:.
- *K a* (for *k*onstant) is used when a field has a type that falls outside the family. In our case, constants cannot be selected and will not be annotated with position information. They always appear on the left-hand side of a :∗:.
- *U* is the [ ] of the list of constructors. [4]
- *pf* :>: *ix* tags the pattern functor *pf* as representing datatype *ix*.

Using these building blocks PF_{Tuples} completely mirrors the structures of the *Expr* and *Type* datatypes. The building blocks' definitions are chosen such that the pattern functor and *Either Expr Type* are isomorphic, i.e. values of one type can be translated to corresponding values of the other type. This fact is reflected by MultiRec's *Fam* type class:

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

The first function, *from*, translates from values of the original two datatypes to the pattern functor versions while the second function, *to* takes care of the reverse. These functions' types introduce a few new things.

The first of these is their first argument, *φ ix*. In our example *φ* is the family *Tuples*:

```
data Tuples :: ∗ → ∗where
  Expr :: Tuples Expr
  Type :: Tuples Type
```

Type *Tuples* is a generalized algebraic datatype (GADT) that enumerates all the datatypes in the family, with one constructor for each datatype. Each constructor indexes *Tuples* with the type it represents. When *from* pattern matches on one of these constructors, the polymorphic type argument *ix* is refined to a specific, concrete datatype after which case analysis can be done on the second argument to produce a value of the pattern functor.

The dual *to* also receives a *Tuples* value, after which it can pattern match on a value of the pattern functor to produce a value of one of the original types.

[4]In the MultiRec paper, *U* is only used to describe constructors with zero fields and does not appear in pattern functors of constructors that do have fields. However, treating constructors consistently as cons lists allows for more elegant type functions later on.

MultiRec abstracts over specific pattern functors through a type synonym family *PF*, which appears in the type signatures of *from* and *to*:

$$\textbf{type}\ \textit{family PF}\ \phi :: (* \to *) \to * \to *$$

Again, in our specific example, ϕ stands for *Tuples*, and we have already seen our example's pattern functor, so writing an instance of this type family is easy:

$$\textbf{type instance}\ \textit{PF Tuples} = \textit{PF}_{\textit{Tuples}}$$

If *PF Tuples* has kind $(* \to *) \to * \to *$, it must receive two type arguments before it becomes a concrete type. From **class** *Fam* we learn that they are called *r* and *ix*. These arguments are shared by each functor building block, all the way down to the leaves.

The latter argument *ix* indicates the original type the pattern functor represents. For example, *PF Tuples r Type*, for some *r*, is the type of a pattern functor value representing a *Type*. Functions *from* and *to* use this to restrain their output and input functors appropriately.

The *r* serves a purpose similar to that in *ExprF r*: it is used to control the values at the recursive child positions. But where the *r* in *ExprF r* had kind $*$, this *r* has kind $* \to *$: *r*'s argument indicates which type is recursed into. This argument was unnecessary in the previous chapter, because families of datatypes were not supported yet and there was only one type to recurse into.

Let us take a look at the definitions of the building blocks so that we can write an **instance** *Fam Tuples*.

```
data (f :+: g) r ix = L (f r ix) | R (g r ix)
infixr 5 :+:
```

The two constructors reflect the choice between the *L*eft functor and the *R*ight functor. The two type arguments *r* and *ix* are simply passed down to either functor.

```
data (f :*: g) r ix = (f r ix) :*: (g r ix)
infixr 7 :*:
```

Values of the product type constructor contain one value of either side. Again, the type arguments are passed on unchanged.

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

The *Tag* constructor fixes the index *ix* of the enclosed functor to be the same as the right-hand side of the :>: type constructor. This ensures that when creating a pattern functor value, only values of the proper branch representing a particular datatype are allowed, and—conversely—that only values of the right branch need to be pattern matched on when converting the pattern functor values back to the original datatypes.

```
data K a r ix = K a
```

Constants discard the *r* and *ix* and only hold a simple value of type *a*.

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

An *I xi* in a pattern functor marks a recursive occurrence of type *xi*. Recursive positions do not need be the same as the enclosing type, so the *ix* argument is ignored. The *r* determines what is actually done with the *xi*. There are several interesting choices for *r*. Two of them are the following datatypes:

```
newtype I∗ a   = I∗ {unI∗ :: a}
newtype K∗ a b = K∗ {unK∗ :: a}
```

The former results in values of the original datatypes at the recursive positions, while the latter ignores the index and uses a constant type. The former is used in the *Fam* type class, so *to* and *from* effectively transform the values only one level deep (a *shallow* conversion). We will see another example of *r* later on when we look at fixed points of pattern functors.

There is also a building block that describes constructor metadata, such as its name or fixity, but we will not need that information and it is safe to leave it out of our pattern functor, so we omit the description of its workings here.

Now that we know how the building blocks are defined, we can write the conversion functions for our *Tuples* family. First conversion from the original datatypes to the pattern functors:

```
instance Fam Tuples where
  from Expr ex   = L ∘ Tag $ case ex of
    EAdd x y   → L             $ I (I∗ x) :∗: I (I∗ y) :∗: U
    EMul x y   → R ∘ L         $ I (I∗ x) :∗: I (I∗ y) :∗: U
    ETup x y   → R ∘ R ∘ L     $ I (I∗ x) :∗: I (I∗ y) :∗: U
    EIntLit n  → R ∘ R ∘ R ∘ L $ K n      :∗: U
    ETyped e t → R ∘ R ∘ R ∘ R $ I (I∗ e) :∗: I (I∗ t) :∗: U
  from Type ty   = R ∘ Tag $ case ty of
    TyInt      → L             $ U
    TyTup x y  → R             $ I (I∗ x) :∗: I (I∗ y) :∗: U
```

The outermost case analysis shows the magic that is pattern matching on GADT values: both *Expr* and *Type* are constructors of the *Tuples* GADT, and depending on which value is given to *from*, the inner case analyses can pattern match on constructors of the relevant, more specific types.

The two outer case constructs wrap all of their results in $L \circ Tag$ and $R \circ Tag$, respectively. This matches with the pattern functor's outer layers: PF_{Expr} :>: *Expr* :+: PF_{Type} :>: *Type*.

Type PF_{Expr} uses nested binary sums. Because these associate to the right, the constructors progressively need more *R*'s in *from*'s definition to reach their position in the pattern functor. To the right of the dollar signs is the construction of each constructor's fields.

The opposite *to* is very similar to *from*, except that the pattern functor constructors are on the left-hand side and the original datatype's constructors are on the right. Unfortunately we cannot eliminate some parentheses like we could in *from* using function composition.

```
to Expr (L (Tag ex)) = case ex of
  L            (I (I_* x) :*: I (I_* y) :*: U)      → EAdd x y
  R (L         (I (I_* x) :*: I (I_* y) :*: U))     → EMul x y
  R (R (L      (I (I_* x) :*: I (I_* y) :*: U)))    → ETup x y
  R (R (R (L   (K n :*: U))))                       → EIntLit n
  R (R (R (R (I (I_* x) :*: I (I_* y) :*: U))))     → ETyped x y
to Type (R (Tag ty)) = case ty of
  L            U                                    → TyInt
  R            (I (I_* x) :*: I (I_* y) :*: U)      → TyTup x y
```

Finally, we need to write instances of *El* for every member of our family. This type class is used in certain instances on building blocks when proofs (elements of the GADT *Tuples*) are needed:

```
class El φ ix where
  proof :: φ ix
instance El Tuples Expr where
  proof = Expr
instance El Tuples Type where
  proof = Type
```

That's it! Now we have everything we need to do some serious multi-datatype generic programming.

6.3 Fixed points in MultiRec

As we saw, the pattern functor embedding is shallow. Because the pattern functor abstracts over the recursive positions, we can use fixed points again to achieve a deep embedding: we just need to fix the type argument *r*—which determines the shape of the children—to the fixed point itself. To find out what the fixed point datatype should look like for a higher-order functor *f* (for example, *PF Tuples*), we write down three iterations of the recursion:

```
type InfiniteFix f ix = f (f (f (...))) ix
```

As before, we observe the regularity of the recursion in this infinite datatype to be able to express the recursion in a Haskell datatype:

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

Now a deep embedding of the a value of the *Tuples* can be expressed as *HFix* (*PF Tuples*).

And because we are using fixed points again, we can insert our annotations at every level. We do not need to build a custom *Ann* datatype like before: we can reuse building blocks *K* and :∗: to couple a constant with a functor.

This gives us what we need to build the MultiRec version of *AnnFix* and $AnnFix_1$. We reuse these names here: it will always be clear from the context which is meant, because the two versions of the fixed points are never mixed.

$$\textbf{type}\ AnnFix\ \ x\ \phi = HFix\ (K\ x :\!*\!: PF\ \phi)$$
$$\textbf{type}\ AnnFix_1\ x\ \phi = (PF\ \phi)\ (AnnFix\ x\ \phi)$$

Using *K* means the index is discarded, making the annotation type *x* independent of the index. If we wanted, we could use an associated datatype (CKPJM05) to make the annotation type depend on the specific datatype in the family. But we will be using source locations throughout the whole tree, so *K* suffices.

6.4 Pattern functors are traversable

If we can translate a value to its pattern functor, we can automatically *traverse* it[5]: all we need to do is build traversal functions for the building blocks. What would such a traversal function look like for the generic building blocks? Recall the traversal function described by McBride and Paterson:

$$traverse :: (Traversable\ t, Applicative\ f) \Rightarrow (a \rightarrow f\ b) \rightarrow t\ a \rightarrow f\ (t\ b)$$

The first argument $a \rightarrow f\ b$ is used on the recursive positions of a datatype constructor. The resulting actions are then sequenced using $(\circledast)$, resulting in one bigger action with the new traversable datatype as result.

In the higher-order version of *traverse*, the first function will also be called with the recursive positions as arguments. However, in the generic case, we do not know the type of the recursive position: it might be any type in the family ϕ. For this reason, the supplied function must be polymorphic in its index so that we can give it a child of any family type. To help the programmer that writes this function, we also pass it a witness $\phi\ ix$ so that the function can pattern match on this witness, refining *ix* to a known type.

In *traverse* the first argument may also change the child type from *a* to *b*. We cannot allow this in the generic version: the index *ix* must remain the same, because a constructor cannot suddenly change the types of its children. We can however change the recursive type constructor *r*.

All these considerations are reflected in MultiRec's version of *Traversable*, called *HFunctor*:

$$\begin{aligned}&\textbf{class}\ HFunctor\ \phi\ f\ \textbf{where}\\ &\quad hmapA :: Applicative\ a \Rightarrow\\ &\qquad (\forall xi.\phi\ xi \rightarrow r\ xi \rightarrow a\ (r'\ xi)) \rightarrow \phi\ ix \rightarrow f\ r\ ix \rightarrow a\ (f\ r'\ ix)\end{aligned}$$

[5]Starting with GHC 6.12, datatypes may derive *Traversable* instances!

The first argument is the action that is applied to every child position. The type guarantees that it works for all indices and that the action does not change the index of a specific child.

The presence of *hmapA*'s second argument, a witness of type $\phi\ ix$, makes it much easier to fix the type argument ϕ when calling hmapA.

Just like in the *Data.Traversable* module, we can write monadic and pure versions of *hmapA*, making use of a (trivial) **instance** *Applicative* I_*:

```
hmapM :: (HFunctor φ f, Monad m) ⇒
  (∀xi.φ xi → r xi → m (r′ xi)) → φ ix → f r ix → m (f r′ ix)
hmap    :: HFunctor φ f ⇒
  (∀xi.φ xi → r xi → r′ xi)        → φ ix → f r ix →      f r′ ix
hmapM f p x = unwrapMonad (hmapA (λp_i x → WrapMonad (f p_i x)) p x)
hmap   f p x = unI_*           (hmapA (λp_i x → I_*              (f p_i x)) p x)
```

All that rests is to show that these functions can actually be implemented on pattern functors. Again, we start with the building blocks of individual constructors:

```
instance HFunctor φ U where
  hmapA _ _ U         = pure U
instance HFunctor φ (K x) where
  hmapA _ _ (K x)     = pure (K x)
instance El φ xi ⇒ HFunctor φ (I xi) where
  hmapA f _ (I x)     = I      <$> f proof x
```

The most interesting one is *I*, where f is used on the recursive position. That instance also explains why ϕ is part of the class header: a proof for the recursive position must be produced to be able to call *f* on the child.

The instances for the combinators use applicative style to traverse child functors:

```
instance (HFunctor φ f, HFunctor φ g) ⇒ HFunctor φ (f :+: g) where
  hmapA f p (L x)     = L     <$> hmapA f p x
  hmapA f p (R y)     = R     <$> hmapA f p y
instance (HFunctor φ f, HFunctor φ g) ⇒ HFunctor φ (f :*: g) where
  hmapA f p (x :*: y) = (:*:) <$> hmapA f p x ⊛ hmapA f p y
instance HFunctor φ f ⇒ HFunctor φ (f :>: ix) where
  hmapA f p (Tag x)   = Tag   <$> hmapA f p x
```

6.5 Error catamorphisms in MultiRec

The next step is to translate the error catamorphisms we discussed in section 5.2 to the 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. We just need to take into account the higher-order recursion parameter *r* and the extra index *ix* of the pattern functor:

$$\textbf{type}\ \mathit{ErrorAlg}_{\mathit{PF}}\ f\ e\ a = \forall ix. f\ (K_*\ a)\ ix \rightarrow \mathit{Either}\ e\ a$$

Here we choose $K_*\ a$ for *r*, saying that the children should always contain a value of type *a*, regardless of their index. 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 pattern functor version of *errorCata*:

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

Although the names of the constructors that are pattern matched on are different, the function's implementation is similar to the old *errorCata*. An extra proof must be carried around and passed to *hmapA*, but just as before the *Applicative* instance of *Except* is used to collect errors.

However, writing such an algebra for this catamorphism is not as easy as previously, because the algebra cannot use the constructors of the original datatype. Instead, it has to pattern match on the chains of *L*'s and *R*'s of the pattern functor, which make the algebra a lot longer and less clear.

The same problem occurs when writing normal algebras (as opposed to error algebras). The MultiRec paper (RHLJ09) solves the problem by automatically translating the pattern functor to a more convenient type in which the algebras can be expressed, using a type synonym family. This derived type cannot be applied directly to values of the pattern functor, but it is translated back to a function that is applicable to pattern functors using a type class with instances for the various functor building blocks.

We will adapt this strategy to work for our error catamorphisms. We skip the discussion of normal catamorphisms; the reader is referred to the MultiRec paper for this. Instead, we jump directly to the encoding of the error catamorphisms in MultiRec. It all starts with the type synonym family that computes the more convenient type:

```
type family ErrorAlg
  (f :: (* → *) → * → *)   -- pattern functor
  (e :: *)                 -- error type
  (a :: *)                 -- result type
   :: *                    -- resulting algebra type
```

Since this type function will be applied to pattern functor building blocks, we need to give an instance for every single building block. Let us start with the building blocks of individual constructors:

```
type instance ErrorAlg U          e a = Either e a
type instance ErrorAlg (K b :*: f) e a = b → ErrorAlg f e a
type instance ErrorAlg (I xi :*: f) e a = a → ErrorAlg f e a
```

The first case handles the base case of every constructor: for every constructor, the error algebra should end in *Either e a*. The second and third cases handle cons nodes of constant fields and recursive fields, introducing an extra argument to the algebra. For constant fields, the type of this extra argument is equal to the type *b* of the field. For recursive positions, however, we ignore the index that is recursed on and expect an argument whose type equals the result type of the algebra. So just like before, the algebra may assume that children have already been folded and that there were no errors. Again, if desirable, the algebra's result type can be made dependent on the index using a associated datatype.

```
type instance ErrorAlg (f :+: g)  e a = (ErrorAlg f e a, ErrorAlg g e a)
type instance ErrorAlg (f :>: xi) e a = ErrorAlg f e a
```

Constructors in a datatype and datatypes in a family are combined using :+:. In such cases, algebras have to be provided for both options, so the algebras are coupled together in a tuple. Tags are ignored; the recursive algebra type is used directly.

With these instances, the computed types *ErrorAlg* PF_{Expr} and *ErrorAlg* PF_{Type} are equivalent to these type synonyms:

```
type ExprErrorAlg e a
  =   (a → a → Either e a)   -- EAdd
  :&: (a → a → Either e a)   -- EMul
  :&: (a → a → Either e a)   -- ETup
  :&: (Int   → Either e a)   -- EIntLit
  :&: (a → a → Either e a)   -- ETyped
type TypeErrorAlg e a
  =              Either e a   -- TyInt
  :&: (a → a → Either e a)   -- TyTup
```

Here (:&:) = (,), a right-associative infix notation for binary tuples, is used to make the types easier to read.

As said before, these derived algebra types are very convenient to work with, but they cannot be directly applied to pattern functors. To still be able to use them, we describe how to convert such a convenient algebra to one that can work on pattern functors using a type class:

```
class MkErrorAlg f where
  mkErrorAlg :: ErrorAlg f e a → ErrorAlg_PF f e a
```

Instances of this class correspond directly with the instances of the type synonym family above:

```
instance MkErrorAlg U where
  mkErrorAlg x          U              = x
instance MkErrorAlg f ⇒ MkErrorAlg (K a :*: f) where
  mkErrorAlg alg        (K x :*: f)    = mkErrorAlg (alg x) f
instance MkErrorAlg f ⇒ MkErrorAlg (I xi :*: f) where
  mkErrorAlg alg        (I (K* x) :*: f) = mkErrorAlg (alg x) f
instance MkErrorAlg f ⇒ MkErrorAlg (f :>: xi) where
  mkErrorAlg alg        (Tag f)        = mkErrorAlg alg f
instance (MkErrorAlg f, MkErrorAlg g) ⇒ MkErrorAlg (f :+: g) where
  mkErrorAlg (alg_f, _) (L x)          = mkErrorAlg alg_f x
  mkErrorAlg (_, alg_g) (R y)          = mkErrorAlg alg_g y
```

Here the benefit we alluded to in footnote 4 becomes apparent: because lists of field types always end in the nil type *U*, we don't have to write instances for lone occurrences of *K a* and *I xi* as they always appear on the left-hand side of a (:*:).

To demonstrate that this approach works in practice, we look at an algebra that infers the type of expressions of our Tuples language.

```
inferType :: ExprErrorAlg String Type :&: TypeErrorAlg String Type
inferType = (equal "+" & equal "*" & tup & const (Right TyInt) & equal "::")
        & (Right TyInt & tup)
  where
    equal op ty1 ty2
      | ty1 ≡ ty2 = Right ty1
      | otherwise = Left ("lhs and rhs of " ++ op ++ " must have equal types")
    tup ty1 ty2     = Right (TyTup ty1 ty2)
```

The algebra says that both operands of $+$ and $*$ must have equal types. The algebra also checks whether explicit type signatures using :: match the types of the expressions on the left-hand sides.

To test this algebra, we use a function *readExpr* :: *String* → *AnnFix Bounds Tuples Expr* we will define later when we discuss parsing annotated expressions with MultiRec:

```
> let expr1 = readExpr "(1, (2, 3))"
> errorCata (mkErrorAlg inferType) Expr expr1
OK (TyTup TyInt (TyTup TyInt TyInt))
> let expr2 = readExpr "(1 :: (Int, Int), 2 + (3, 4))"
> errorCata (mkErrorAlg inferType) Expr expr2
Failed
  [("lhs and rhs of :: must have equal types",
    Bounds {leftMargin = (1,1), rightMargin = (16,16)})
  ,("lhs and rhs of + must have equal types",
    Bounds {leftMargin = (17,18), rightMargin = (28,28)})
  ]
```

6.6 Constructing recursively annotated trees

In section 6.3 we translated the *AnnFix* and $AnnFix_1$ type synonyms from section 5 to use MultiRec terms. Function *mkAnnFix* can be translated in a similar fashion:

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

When we developed the parser combinators, we produced $AnnFix_1$s by taking one of *ExprF*'s constructors and giving it fully annotated *AnnFix* children. But producing an $AnnFix_1$ in the MultiRec version is not as easy: we cannot use the constructors of the original datatypes *Expr* and *Type* and give them annotated children, because the constructors' types only allow unannotated values as fields. Instead, to make 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 to write, and that makes it easier to make mistakes when writing them down. They have to be written for each possible constructor, and then they have to be used during parsing instead of the real constructors. It would be much nicer if we could find a way to avoid this.

The way we will solve this problem in this thesis is by making 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, when parsing the sentence `"2 + 3"`, first the 2 is parsed, then the 3 and only then their sum is constructed and returned.

The parser maintains an explicit stack of fully annotated trees. The stack contains exactly those trees that will form the children of later nodes, until the very end when it will contain the final result of the parser. In the example above, the parser would first push *EIntLit* 2, then *EIntLit* 3 and finally *EAdd* (*EIntLit* 2) (*EIntLit* 3).

Every time an unannotated node is pushed onto the stack, the parser supplies the annotation for the root of the node. Because the node's children and their annotations will already be present on the stack, the parser has enough information at that moment to form a new, fully annotated tree. Let us call the action of pushing an element onto the stack *yield*. Then the construction of the annotated parse tree for `"2 + 3"` looks like this:

```
do
  n2 ← yield Expr (Bounds {leftMargin = (0,0), rightMargin = (1,2)}) (EIntLit 2)
```

$n3 \leftarrow \mathit{yield}\ \mathit{Expr}\ (\mathit{Bounds}\ \{\mathit{leftMargin} = (3,4), \mathit{rightMargin} = (5,5)\})\ (\mathit{EIntLit}\ 3)$
$n5 \leftarrow \mathit{yield}\ \mathit{Expr}\ (\mathit{Bounds}\ \{\mathit{leftMargin} = (0,0), \mathit{rightMargin} = (5,5)\})\ (\mathit{EAdd}\ n2\ n3)$
return n5

The first argument to *yield* is a witness of the *Tuples* GADT, indicating the type of the expression that is being pushed onto the stack. The second argument is the annotation that goes with the root of the tree, while the third and final argument is the tree itself.

Note that there is some overlap in the information that is pushed onto the stack. First 2 and 3 are pushed individually, and then in the third *yield* they are given to the stack again as children of the *EAdd* node. If the integer literals are given twice, which instances does *yield* use? The answer is that *yield* only uses the root constructors of the elements given to it. The children of the *EAdd* node in the third call to *yield* are discarded, so yielding *EAdd* $\perp$ $\perp$ would have worked just as well. This once again shows that redundant information is bad, but it is the price we pay for being able to use the original datatype constructors rather than the pattern functor versions. Alternative solutions to the problem are discussed in chapter 7.

Let us take a closer look at the exact steps *yield* takes whenever it is called:

1. A new element is being pushed onto the stack. We have been given the accompanying witness of type $\phi\ ix$, so we can do a shallow conversion to the pattern functor version of the constructor using *from* in type class *Fam*. The resulting value is of type $PF\ \phi\ I_*\ ix$.
2. Figure out how many children the constructor has. We can express this computation in terms of *hmapA*, traversing the constructor's pattern functor and counting the child positions. Let us call the number of children *n*.
3. Pop *n* children from the stack. Every child is fully annotated (by induction; read on) and has type $\mathit{AnnFix}\ x\ \phi\ xi$, for some index *xi*.
4. Distribute the *n* children over the pattern functor of the constructor being pushed. This results in a node which has fully annotated children but still lacks a top-level annotation, i.e. a value of type $\mathit{AnnFix}_1\ x\ \phi\ ix$.
5. Using *mkAnnFix*, tie the annotation that was passed to *yield* to the AnnFix_1 value, producing a value of $\mathit{AnnFix}\ x\ \phi\ ix$.
6. Finally, push this new, fully annotated tree onto the stack.

A consequence of pushing redundant information is that some of the steps outlined above might fail. Step 3 might find there are not enough (fewer than *n*) children on the stack if a previous parse operation neglected to *yield* its results. Step 4 might find it is trying to replace unannotated children by annotated values of the wrong type. For example, *EAdd* expects two *Expr* children, but the previous yield statements might have pushed *Type*s onto the stack. Finally, at the end of the parsing, we expect exactly one tree in the stack: another thing that might not be the case.

The good news is that the *yield*ing of values can be hidden in standard combinators such as *unit*, *chainl* and *chainr*. This means that correct use of *yield* only has to be verified for these combinators and that the final parser cannot do anything wrong if it is expressed in terms of these combinators. In fact, as we will see in a bit, the parser for the *Expr* part of the *Tuples* family is surprisingly similar to the parsers defined in section 5.3.

Before we look at these parsing combinators, however, let us look at the code behind *yield*. We can separate the parsing from the *yield*ing by defining a new monad transformer for the *yield*ing process and then later stacking the parser transformer with the *yield* transformer. As is common when defining a new transformer, we capture the primitive operations in their own type class:

```
class Monad m ⇒ MonadYield m where
  type Φ_Yield    m :: ∗ → ∗
  type AnnType m :: ∗
  yield             :: Φ_Yield m ix → AnnType m → ix → m ix
```

Class *MonadYield* needs only one primitive operation, but it uses two associated types, expressed using a type synonym family. By putting these functions inside the class, we express the fact that instances of these type functions go hand in hand with instances of the class itself.

What type do the stack elements have? The elements might be of different types within a family. In order to be able to put them in the same list, we can existentially quantify over the family indices. By storing them together with proof values of the family GADT, we can always recover the indices again later by pattern matching on the proof. We define a new datatype to this end and call it *AnyF*, because it can store an *f* indexed by any type in a family ϕ:

```
data AnyF φ f where
  AnyF :: φ ix → f ix → AnyF φ f
type AnyAnnFix x φ = AnyF φ (AnnFix x φ)
```

The first argument to the *AnyF* constructor is the proof; the second is an *f* indexed by the type indicated by the proof. In our case, we set *f* to be *AnnFix x* ϕ and call such an element *AnyAnnFix x* ϕ. Now we can write an instance of *MonadYield*:

```
newtype YieldT x φ m a = YieldT (StateT [AnyAnnFix x φ] m a)
  deriving (Functor, Monad)
instance MonadTrans (YieldT x φ) where
  lift = YieldT ∘ lift
instance (Monad m, HFunctor φ (PF φ), EqS φ, Fam φ) ⇒
  MonadYield (YieldT x φ m) where
  type Φ_Yield    (YieldT x φ m) = φ
  type AnnType (YieldT x φ m) = x
  yield                          = doYield
```

In *YieldT x* ϕ *m a*, *x* is the type of the annotation (such as *Bounds*), ϕ is the family GADT (such as *Tuples*), *m* is the underlying monad and *a* is the return value of the monad. We keep track of the stack by making *YieldT* a **newtype** wrapper around a *StateT* monad with a list of *AnyAnnFix*es as state. As any good monad transformer, a *MonadTrans* instance is supplied. The type functions Φ_{Yield} and *AnnType* point to the relevant type arguments. The definition of *doYield* is postponed; we will look at it in a moment. First, we need to define the *EqS* class constraint.

When we distribute the annotated children over a pattern functor, we need to make sure that the children are of the right type. We will use hmapA to perform the distribution, and the transformation function passed to *hmapA*, which has type $\forall xi.\phi\ xi \rightarrow r\ xi \rightarrow a\ (r'\ xi)$, reflects the fact that the new children should have the right index.

The transformation function is passed a witness of type $\phi\ xi$ and the old child of type $r\ xi$, and the resulting action should return a new child of type $r'\ xi$, i.e. with the same index. How can we make sure that the element we removed from the stack has the right index? This is what *EqS* is for:

```
class EqS φ where
  eqS :: φ ix₁ → φ ix₂ → Maybe (ix₁ :=: ix₂)
data (:=:) :: ∗ → ∗ → ∗ where
  Refl :: ix :=: ix
```

Constructor *Refl* can only be constructed if the two type operands to (:=:) are equal. As a result, if two type variables ix_1 and ix_2 are in scope, pattern matching on the *Refl* constructor of type $ix_1 :=: ix_2$ tells the Haskell compiler that these two type variables must be equal. This explains the return type of *eqS*: given two witnesses, if they turn out to be one and the same witness, we do not just want the result *True* (as is the case with the normal equality operator $(\equiv)$), but we also want a proof that $ix_1 :=: ix_2$. We can use *eqS* when distributing the children: compare the witness argument to *hmapA*'s first argument to the witness that is stored in the *AnyAnnFix*. If they are equal, then we can convince the compiler that it is okay to return the annotated child.

```
doYield :: (Monad m, HFunctor φ (PF φ), EqS φ, Fam φ) ⇒
  φ ix → x → ix → YieldT x φ m ix
doYield p bounds x = YieldT $ do
  let pfx =  from p x
  let n   =  length (children p pfx)
  stack  ←  get
  if length stack < n
    then
      error $ "structure mismatch: required " ++ show n ++
            " accumulated children but found only " ++
            show (length stack)
    else do
      let (cs, cs′)    = splitAt n stack
      let newChild = evalState (hmapM distribute p pfx) (reverse cs)
      put (AnyF p (HIn (K bounds :∗: newChild)) : cs′)
      return x
distribute :: EqS φ ⇒ φ ix → I∗ ix → State [AnyAnnFix x φ] (AnnFix x φ ix)
distribute p₁ _ = do
```

```
    xs ← get
    case xs of
      [] →
        error "structure mismatch: too few children"
      AnyF p₂ x : xs′ →
        case eqS p₁ p₂ of
          Nothing → error "structure mismatch: incompatible child type"
          Just Refl → do put xs′; return x
```

The code closely follows the steps outlined earlier. One thing to note is that because *doYield* uses *splitAt* to pop the children, the resulting list has to be reversed: the child that was pushed last appears at the front of the stack, while we want that child to be in the last position of the list we give to *distribute*. That *distribute* ignores the unannotated children and simply replaces them is made explicit by the underscore as its second argument.

6.7 Parsing expressions

In the previous section we alluded to hiding the calls to *yield* in parser combinators. Now that we have fully defined the *YieldT* monad transformer, we can build a monad stack that allows yielding while parsing and actually define these parser combinators. In section 5.3 we defined a type synonym *P*; we will reuse *P* and extend it to define a type synonym *YP* that incorporates *YieldT*:

```
type YP s φ m = P s (YieldT Bounds φ m)
```

Why should the monads be stacked in this way? Stacked like this, we can use the choice parser combinator (<|>) to define alternatives in the grammar. If *YieldT* were on the outside, we would only be able to use choice in the inner monad *P*, from where we cannot *yield* any values. On the other hand, now that *P* is on the outside we can express choice and still have access to *yield* from within the alternatives.

We have to be careful that while evaluating the choices, no *yield* side effects take place. Parsec partially solves this by specifying that p_1 <|> p_2 first tries p_1 and does not consider p_2 anymore if p_1 has consumed any input. Since we only *yield* values after consuming some input, we are safe. However, Parsec also offers the *try* combinator which lifts this restriction and allows arbitrary backtracking. If *try* is used, we have no safety guarantees anymore. For that reason, we should be very careful to use *try*, or—better yet—not use it at all. Fortunately, we will not need it in the definition of our example *Tuples* parser.

Let us take a look at the *unit* parser combinator we have seen in section 5.3, translated to the YP monad stack:

```
unit :: (Fam φ, EqS φ, HFunctor φ (PF φ), Monad m) ⇒
  φ a → YP s φ m a → YP s φ m a
unit w p = do
  left ← getPos
  x    ← p
  mkBounded w left x
mkBounded :: (Fam φ, EqS φ, HFunctor φ (PF φ), Monad m) ⇒
  φ a → Range → a → YP s φ m a
mkBounded w left x = do
  right ← getPos
  lift $ yield w (Bounds left right) x
```

Although the types are different and more involved, the definition of *unit* is identical to that of the original *unit*, with the exception of an extra witness parameter of type *φ a*. Only *mkBounded*'s definition has changed significantly: instead of returning the result of a call to *mkAnnFix*, it calls *yield*.

The other two combinators, *chainl* and *chainr*, are also the same as before, again with the exception of the witness parameter:

```
chainr :: (Fam φ, EqS φ, HFunctor φ (PF φ), Monad m, Show a) ⇒
  φ a → YP s φ m a → YP s φ m (a → a → a) → YP s φ m a
chainr w px pf = fix $ λloop → do
  left ← getPos
  x    ← px
  option x $ do
    f ← pf
    y ← loop
    mkBounded w left (f x y)
chainl :: (Fam φ, EqS φ, HFunctor φ (PF φ), Monad m, Show a) ⇒
  φ a → YP s φ m a → YP s φ m (a → a → a) → YP s φ m a
chainl w px pf = do
    left ← getPos
    px ≫= rest left
  where
    rest left = fix $ λloop x → option x $ do
      f ← pf
      y ← px
      mkBounded w left (f x y) ≫= loop
```

The actual parser for the *Tuples* family of datatypes is unsurprising, but we include it here for the sake of symmetry with the previous section. As before, we do not include the code for the lexer. The lexer token constructor names can be recognized by a prefix *T*.

```
type ExprParser = YP ExprToken Tuples Identity
pExpr      :: ExprParser Expr
pExpr      = do
  left ← getPos
  ex   ← pAdd
  option ex $ do
    pToken TDoubleColon
    ty ← pType
    mkBounded Expr left (ETyped ex ty)
```

```
pAdd      :: ExprParser Expr
pAdd      = chainl Expr pMul    (EAdd <$ pToken TPlus)
pMul      :: ExprParser Expr
pMul      = chainl Expr pFactor (EAdd <$ pToken TStar)
pFactor   :: ExprParser Expr
pFactor   = pIntLit <|> pTupleVal
pIntLit   :: ExprParser Expr
pIntLit   = unit Expr $ (λ(TIntLit n) → EIntLit n) <$> satisfy isIntLit
pTupleVal :: ExprParser Expr
pTupleVal = pTuple Expr pExpr ETup
pType     :: ExprParser Type
pType     = pTyInt <|> pTyTuple
pTyInt    :: ExprParser Type
pTyInt    = unit Type $ TyInt <$ pToken TInt
pTyTuple  :: ExprParser Type
pTyTuple  = pTuple Type pType TyTup
pTuple    :: Tuples ix → ExprParser ix → (ix → ix → ix) → ExprParser ix
pTuple w pEl f = do
  left ← getPos
  pToken TLParen
  lhs ← pEl
  ty  ← option lhs $ do
    pToken TComma
    rhs ← pEl
    mkBounded w left (f lhs rhs)
  pToken TRParen
  return ty
```

6.8 Annotation-guided exploring

In contrast with the annotation-guided exploring in section 5, pattern functors give us enough information to define genuine zippers. An implementation of such a zipper is available in package `zipper` on Hackage[6].

We will discuss the API of the zipper, but not its implementation. For details on the implementation, the reader is referred to the MultiRec paper (RHLJ09). The current version of the zipper (`0.3`) is intended to work on shallowly converted datatypes. In order to be able to use it for recursively annotated trees, a slight modification has been made to the library. The modified version is listed in Appendix A.

An overview of the API is listed in figure 6.1. The names of the arguments of *FixZipper* are consistent with those in previous datatypes: ϕ is the family GADT, f is the pattern functor over which the zipper is computed and *ix* is the top-level index.

[6] `http://hackage.haskell.org/package/zipper`

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

Figure 6.1: The API of the MultiRec-based fixpoint zipper.

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 and returns the tree that was originally *enter*ed. Because a zipper value implicitly contains a witness *φ ix*, functions that *accept* a zipper (such as *leave*) do not need an extra witness argument.

The navigation functions *down*, *up*, *right* and *left* all have the same type, which—as before—has been called *Nav*. And just as before, navigation steps can be composed using ⋙.

Finally, *on* takes a function, applies it to the current focus and then returns the result.

If we instantiate a *FixZipper* with *K x* :∗: *PF φ* for *f*, it will hold 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*:

```
focusAnn :: AnnZipper φ x ix → x
focusAnn = on (\_(HIn (K x :∗: _)) → x)
```

The heart of the code in section 5.4 was the *explore* function. Most other functions were expressed in terms of *explore*. The function's generic sibling is very similar:

```
explore :: Zipper φ (PF φ) ⇒
  φ ix → (x → ExploreHints) → (AnnFix x φ) ix → [AnnZipper φ x ix]
explore p hints = explore′ hints ∘ enter p
explore′ :: Zipper φ (PF φ) ⇒
  (x → ExploreHints) → AnnZipper φ x ix → [AnnZipper φ x ix]
explore′ hints root = [z | (dirOk, zs) ← dirs, dirOk (hints x), z ← zs]
  where
    x = focusAnn root
    dirs =
      [(matchHere,    [root])
      ,(exploreDown,  exploreMore (down root))
      ,(exploreRight, exploreMore (right root))
      ]
    exploreMore = maybe [] (explore' hints)
```

Its type is more complicated, and it receives a witness. The extraction of the annotation is done using *focusAnn* rather than direct pattern matching.

The other functions are also very similar. Again, the type changes and some witnesses are passed, but their essences are the same.

```
findLeftmostDeepest :: (Zipper φ (PF φ)) ⇒
  φ ix → (x → Bool) → AnnFix x φ ix → Maybe (AnnZipper φ x ix)
findLeftmostDeepest p down = listToMaybe ∘ reverse ∘ explore p hints
  where
    hints x
       | down x    = ExploreHints True  True  False
       | otherwise = ExploreHints False False True
selectByRange :: Zipper φ (PF φ) ⇒
  φ ix → Range → AnnFix Bounds φ ix → Maybe (AnnZipper φ Bounds ix)
selectByRange p range@(left, _) = listToMaybe ∘ reverse ∘ explore p hints where
  hints bounds@(Bounds _ (ir, _)) =
      ExploreHints
        {matchHere    = range ‘rangeInBounds‘ bounds
        , exploreDown  = range ‘rangeInRange‘ outerRange bounds
        , exploreRight = left ⩾ ir
        }
selectByPos :: (Zipper φ (PF φ)) ⇒
  φ ix → Int → AnnFix Bounds φ ix → Maybe (AnnZipper φ Bounds ix)
selectByPos p pos = findLeftmostDeepest p (posInRange pos ∘ innerRange)
repairBy :: (Ord dist, HFunctor φ (PF φ)) ⇒
  φ ix → (Range → Range → dist) → AnnFix Bounds φ ix → Range → Bounds
repairBy p cost tree range =
  head (sortOn (cost range ∘ innerRange) (allAnnotations p tree))
repair :: HFunctor φ (PF φ) ⇒
  φ ix → AnnFix Bounds φ ix → Range → Bounds
repair p = repairBy p distRange
moveSelection :: Zipper φ (PF φ) ⇒
  φ ix → AnnFix Bounds φ ix → Nav → Range → Maybe Bounds
moveSelection p tree nav range = focusAnn <$> (selectByRange p range tree ≫= nav)
```

Chapter 7

Future work

7.1 Dealing with AST mutations

The proposal for this thesis contained figure 7.1, depicting the refactoring cycle: let us say the programmer is working in his or her favorite IDE, selects a piece of source code and performs a refactoring action. To do this, the IDE has to map the text selection to a structural selection on the AST in memory: something we have discussed in detail in the previous chapters. Then the selection must be processed so that the refactoring action is carried out, mutating the AST in the process. Perhaps parts of the code must be deleted or moved to a new position, or perhaps new code has to be created.

Dealing with mutations to ASTs is a very interesting problem we have not addressed yet, and it would make a nice subject for further research. For example, how do we make sure the text positions stored in the AST stay correct after a mutation? 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.

But this introduces a whole new set of questions: Can we generically add

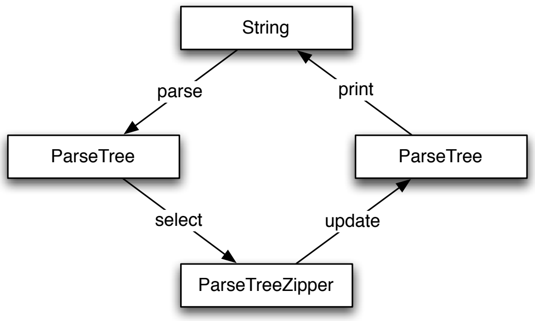


Figure 7.1: The refactoring cycle.

source code to trees? Of course we can put it in an annotation, but how do we do that in such a way that there is no redundant information, i.e. such that a tree's annotation does not also store the source code of its child trees? Do we need to create a derived datatype for this? And how can we efficiently compute position information from the source code? How can we elegantly express the generation of new pieces of code? What would the rewriting API look like? Perhaps work on generic rewriting of unannotated trees (NRH$^+$08) is of help here.

7.2 Contexts in algebras

Both approaches discussed in this thesis relied on the programmer to express their data-consuming functions as algebras to catamorphisms in order for their results to be automatically annotated. Most consumers can be expressed in terms of an algebra, because the result type of the algebra is allowed to be a function *context* $\rightarrow$ *result* where *context* can be any context information necessary to compute the result. In the extreme case it can be the zipper context, in which case the algebra has access to the complete input datatype.

A well-known example is the evaluation of arithmetic expressions with variables. The algebra result type is often something like *Environment* $\rightarrow$ *Integer*. When evaluating a variable, the variable is looked up in the environment to see what value it has. In the evaluation of a binding, a key-value pair is added to the environment.

But such result types do not work well in combination with error algebras: the type $\mathit{ErrorAlgebra}\ f\ e\ (c \rightarrow a)$ is equivalent to $f\ (c \rightarrow a) \rightarrow \mathit{Either}\ e\ (c \rightarrow a)$. This is not very convenient: it means the algebra has to decide whether to throw an error without being able to look at the context. Imagine this in the scenario above: an algebra encounters a *Var* node and would like to throw an error if the variable is unbound in the environment, but is unable to do so because no environment is available. A more convenient algebra type would be $f\ (c \rightarrow a) \rightarrow c \rightarrow \mathit{Either}\ e\ a$, where the context is moved out of the *Either* constructor and thus available when deciding to yield a *Left* or a *Right*. It would be interesting to see if *errorCata* could be changed to work with such algebras.

7.3 Indexed base functors

The first approach (base functors) has as drawback that it cannot be used with multiple datatypes, restricting its use to very simple ASTs. The second approach (MultiRec) has as drawback that the original constructors cannot be used in the derived datatype, making it very difficult to come up with a convenient way to write parsers (or producers in general); we chose for the unsafe, stateful *YieldT* monad transformer. Consumers (e.g. the algebras for the catamorphisms) were a little better: the algebra type could be computed from the AST's pattern functor. But this, too, was not perfect as the parts for the various constructors were combined together in nested tuples. Although it is still *safe* (the compiler will complain if, say, the part for one constructor is omitted), the resulting type errors are big and confusing. Can we combine the best parts of these two approaches and create something in which there is only a single set of constructors (e.g. no counterparts composed of *L*s and *R*s), yet allows families of datatypes? The answer seems to be yes.

The MultiRec paper (RHLJ09) in section 4.2 encodes the pattern functor of their example AST directly as a GADT, with the recursive positions made explicit using a type parameter *r*. The *Tuples* example we have been using would look like this:

```
data Tuples r ix where
  ExAdd :: r Expr  → r Expr → Tuples r Expr
  ExMul :: r Expr  → r Expr → Tuples r Expr
  ExTup :: r Expr  → r Expr → Tuples r Expr
  ExInt :: Integer →           Tuples r Expr
  ExTyp :: r Expr  → r Type → Tuples r Expr
  TyInt ::                     Tuples r Type
  TyTup :: r Type  → r Type → Tuples r Type
```

Using this representation, we would have the good properties of both approaches explored in this thesis. It would be interesting to see how all the related ideas (catamorphisms, parsers, selection translations, et cetera) translate to this approach, and to see what problems or new possibilities would arise.

7.4 Lists of children

The example ASTs in this thesis have not used any lists of children. Lists of child nodes are very common in ASTs; it would be interesting to see how these match with the current approaches.

In the base functor approach, it should not be difficult to use them: the Traversable instances are easy to adapt to constructors with list fields, the algebras work well with them, as do the parsers. The current version of MultiRec, however, has no support for them.

The question of how to generically encode *things* of children in pattern functors is an open one, but it should be a lot easier to consider *lists* of children a special case, introducing a new pattern functor building block just for this purpose. How would this affect all the components we have developed?

Chapter 8

Related work

Probably the closest related work is Proxima (Sch04). Proxima is a generic framework for creating structure editors. Programmers have full control over how the presentation is done, and edit operations can be done on the presentation level and get mapped back to the data model. It provides tools for defining parsers, ASTs, and evaluation using attribute grammars.

To find other related work, one can either narrow the problem to only parser technology, or narrow the problem to specific programming languages.

There is a lot of literature about tracking position information in parsers and parser generators. An old, well-known implementation is lex/yacc (LMB92), while a more recent one is ANTLR (Par07). Both and more have also been mentioned in the introduction. And of course we have seen a lot of Parsec in this thesis.

The other option is to zoom in on specific compiler implementations and take a look at how the problem is handled there. The Glasgow Haskell Compiler, for example, uses the *Located* datatype to couple a value with position information:

data *Located e* = *L SrcSpan e*

Then every child in every AST type is wrapped in an *L* manually to keep track of positions.

In object-oriented implementations, the position information is usually stored in a superclass. For example, in the compiler included in the Eclipse Java Development Tools (JDT), every node in the AST derives from *ASTNode* which provides methods `int getStartPosition()` and `int getLength()`.

Also interesting is the *TextEditor* class in the Eclipse Plug-in Framework that allows a programmer to build text editors for arbitrary languages. It provides support for many features typical for text editors, such as syntax highlighting and content outlines. But the modelling of the AST and the parsing of the source code is completely left to the programmer.

Chapter 9

Conclusion

In this thesis we have looked at ways to generically add position information to recursive datatypes. We have looked at ways in which producers and consumers can be written down in such a way that the position information is constructed or consumed automatically.

The solution we have chosen is based on the fixed point view of datatypes. By expressing a recursive type as a fixed point, it is possible to insert position information at every recursive position.

We have implemented this solution in two ways: one using base functors (Chapter 5), where the programmer decides beforehand that the datatype is written with a type parameter for the children, and the other in terms MultiRec (Chapter 6), a library for generic programming for mutually recursive datatypes.

The former implementation provides good type-safety in all aspects: the shape of the annotated datatype was exactly right, the producers can clearly distinguish between annotated and unannotated trees, and the algebras that we have developed can work with trees either with or without annotations. The major drawback of this implementation is that it can only work with a single datatype.

The latter implementation solves the drawback of the first one: the generic programming library MultiRec, with which the second implementation is built, is designed to work with mutually recursive datatypes. By mimicking the family of datatypes with generic pattern functor building blocks, generic functions and datatypes can be built. But this is also the main problem: the generic constructors are not the same as the original ones, making it very difficult for the producer (in our case the parser) to build recursively annotated trees in an elegant way. Rather than ask the programmer to use the generic constructors (comprised of compositions of *L* and *R*), we allowed the original constructors for the price of a stateful construction model using monad transformer *YieldT*.

With this, we have provided the first steps of turning generic position information from a design pattern into a library.

Appendix A

The modified zipper

The following two sections list the modules that make up the modified zipper, based `zipper-0.3` on Hackage[1]. The first module is a subset of the module zipper, mostly unmodified. The second module contains functions for zippers that work specifically on *HFix* fixed points. There is also a version that works on shallow values, where all the original subtrees are original *ix* values and only the context is represented generically; this version is not included here because it is not used in this thesis.

A.1 module Annotations.MultiRec.Zipper

```
  {-# LANGUAGE FlexibleContexts #-}
  {-# LANGUAGE FlexibleInstances #-}
  {-# LANGUAGE GADTs #-}
  {-# LANGUAGE KindSignatures #-}
  {-# LANGUAGE MultiParamTypeClasses #-}
  {-# LANGUAGE RankNTypes #-}
  {-# LANGUAGE TypeFamilies #-}
  {-# LANGUAGE TypeOperators #-}
  {-# LANGUAGE ScopedTypeVariables #-}
module Annotations.MultiRec.Zipper where
import Prelude hiding (last)
import Control.Monad
import Control.Applicative
import Data.Maybe
import Generics.MultiRec.Base
import Generics.MultiRec.HFunctor
data Loc :: (* → *) → ((* → *) → * → *) → (* → *) → * → * where
  Loc :: φ ix → r ix → Ctxs φ f ix r a → Loc φ f r a
```

[1]`http://hackage.haskell.org/package/zipper-0.3`

```
data Ctxs :: (∗ → ∗) → ((∗ → ∗) → ∗ → ∗) → ∗ → (∗ → ∗) → ∗ → ∗ where
  Empty :: Ctxs φ f a r a
  Push :: φ ix → Ctx f b r ix → Ctxs φ f ix r a → Ctxs φ f b r a
data family Ctx f :: ∗ → (∗ → ∗) → ∗ → ∗
data instance Ctx (K a)      b r ix
data instance Ctx U          b r ix
data instance Ctx (f :+: g)  b r ix = CL    (Ctx f b r ix)
                                    | CR    (Ctx g b r ix)
data instance Ctx (f :*: g)  b r ix = C1    (Ctx f b r ix) (g r ix)
                                    | C2    (f r ix) (Ctx g b r ix)
data instance Ctx (I xi)     b r ix = CId   (b :=: xi)
data instance Ctx (f :>: xi) b r ix = CTag (ix :=: xi) (Ctx f b r ix)
data instance Ctx (C c f)    b r ix = CC    (Ctx f b r ix)
instance Zipper φ f ⇒ HFunctor φ (Ctx f b)      where
  hmapA                  = cmapA
instance Zipper φ f ⇒ HFunctor φ (Ctxs φ f b) where
  hmapA f p′ Empty        = pure Empty
  hmapA f p′ (Push p c s) = liftA2 (Push p) (hmapA f p c) (hmapA f p′ s)
instance Zipper φ f ⇒ HFunctor φ (Loc φ f)      where
  hmapA f p′ (Loc p x s)  = liftA2 (Loc p) (f p x) (hmapA f p′ s)
class HFunctor φ f ⇒ Zipper φ f where
  cmapA       :: Applicative a ⇒ (∀ix.φ ix → r ix → a (r′ ix)) →
                 φ ix → Ctx f b r ix → a (Ctx f b r′ ix)
  fill        :: φ b → Ctx f b r ix → r b → f r ix
  first, last :: (∀b.φ b → r b → Ctx f b r ix → a) →
                 f r ix → Maybe a
  next, prev  :: (∀b.φ b → r b → Ctx f b r ix → a) →
                 φ b → Ctx f b r ix → r b → Maybe a
instance El φ xi ⇒ Zipper φ (I xi) where
  cmapA f p (CId prf) = pure (CId prf)
  fill p (CId prf) x    = castId prf I x
  first f (I x)         = return (f proof x (CId Refl))
  last  f (I x)         = return (f proof x (CId Refl))
  next f p (CId prf) x  = Nothing
  prev f p (CId prf) x  = Nothing
instance Zipper φ (K a) where
  cmapA f p void        = impossible void
  fill p void x         = impossible void
  first f (K a)         = Nothing
  last  f (K a)         = Nothing
  next f p void x       = impossible void
  prev f p void x       = impossible void
instance Zipper φ U where
  cmapA f p void        = impossible void
  fill p void x         = impossible void
  first f U             = Nothing
  last  f U             = Nothing
  next f p void x      = impossible void
  prev f p void x      = impossible void
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :+: g) where
  cmapA f p (CL c)     = liftA CL (cmapA f p c)
  cmapA f p (CR c)     = liftA CR (cmapA f p c)
  fill p (CL c) x      = L (fill p c x)
  fill p (CR c) y      = R (fill p c y)
  first f (L x)        = first (λp z → f p z ∘ CL) x
  first f (R y)        = first (λp z → f p z ∘ CR) y
  last  f (L x)        = last  (λp z → f p z ∘ CL) x
  last  f (R y)        = last  (λp z → f p z ∘ CR) y
  next f p (CL c) x    = next (λp z → f p z ∘ CL) p c x
  next f p (CR c) y    = next (λp z → f p z ∘ CR) p c y
  prev f p (CL c) x    = prev (λp z → f p z ∘ CL) p c x
  prev f p (CR c) y    = prev (λp z → f p z ∘ CR) p c y
instance (Zipper φ f, Zipper φ g) ⇒ Zipper φ (f :*: g) where
  cmapA f p (C1 c y)   = liftA2 C1 (cmapA f p c) (hmapA f p y)
  cmapA f p (C2 x c)   = liftA2 C2 (hmapA f p x) (cmapA f p c)
  fill p (C1 c y) x    = fill p c x :*: y
  fill p (C2 x c) y    = x :*: fill p c y
  first f (x :*: y)    = first (λp z c → f p z (C1 c y)) x ‘mplus‘
                         first (λp z c → f p z (C2 x c)) y
  last  f (x :*: y)    = last  (λp z c → f p z (C2 x c)) y ‘mplus‘
                         last  (λp z c → f p z (C1 c y)) x
  next f p (C1 c y) x  = next (λp′ z c′ → f p′ z (C1 c′          y )) p c x ‘mplus‘
                         first (λp′ z c′ → f p′ z (C2 (fill p c x) c′)) y
  next f p (C2 x c) y  = next (λp′ z c′ → f p′ z (C2 x c′)) p c y
  prev f p (C1 c y) x  = prev (λp′ z c′ → f p′ z (C1 c′ y)) p c x
  prev f p (C2 x c) y  = prev (λp′ z c′ → f p′ z (C2 x  c′          )) p c y ‘mplus‘
                         last  (λp′ z c′ → f p′ z (C1 c′ (fill p c y))) x
instance Zipper φ f ⇒ Zipper φ (f :>: xi) where
  cmapA f p (CTag prf c)   = liftA (CTag prf) (cmapA f p c)
  fill      p (CTag prf c) x = castTag prf Tag (fill p c x)
  first   f   (Tag x)        = first   (λp z → f p z ∘ CTag Refl)     x
  last    f   (Tag x)        = last    (λp z → f p z ∘ CTag Refl)     x
  next    f p (CTag prf c) x = next    (λp z → f p z ∘ CTag prf) p c x
  prev    f p (CTag prf c) x = prev    (λp z → f p z ∘ CTag prf) p c x
instance (Constructor c, Zipper φ f) ⇒ Zipper φ (C c f) where
  cmapA f p (CC c)           = liftA CC (cmapA f p c)
  fill      p (CC c) x       = C (fill p c x)
  first   f   (C x)          = first  (λp z → f p z ∘ CC)      x
  last    f   (C x)          = last   (λp z → f p z ∘ CC)      x
  next    f p (CC c) x       = next  (λp z → f p z ∘ CC) p c x
  prev    f p (CC c) x       = prev  (λp z → f p z ∘ CC) p c x
enter :: Zipper φ f ⇒ φ ix → r ix → Loc φ f r ix
enter p x = Loc p x Empty
on :: (∀xi.φ xi → r xi → a) → Loc φ f r ix → a
on f (Loc p x _) = f p x
```

```
update :: (∀xi.φ xi → r xi → r xi) → Loc φ f r ix → Loc φ f r ix
update f (Loc p x s) = Loc p (f p x) s

impossible :: a → b
impossible _ = error "impossible"

castId   :: (b :=: xi) → (r xi → I xi r ix) → (r b → I xi r ix)
castId   Refl = id

castTag  :: (ix :=: xi) → (f r ix → (f :>: ix) r ix) → (f r ix → (f :>: xi) r ix)
castTag  Refl = id
```

A.2 module Annotations.MultiRec.ZipperFix

```
{-# LANGUAGE RankNTypes #-}
module Annotations.MultiRec.ZipperFix where

import Annotations.MultiRec.Zipper
import Generics.MultiRec.HFix

import Prelude hiding (last)
import Data.Maybe

type FixZipper φ f = Loc φ f (HFix f)

type Nav = ∀φ f ix.Zipper φ f ⇒ FixZipper φ f ix → Maybe (FixZipper φ f ix)

down, down', up, right, left :: Nav

down  (Loc p (HIn x) s)       = first (λp' z c → Loc p' z (Push p c s)) x
down' (Loc p (HIn x) s)       = last  (λp' z c → Loc p' z (Push p c s)) x
up    (Loc p x Empty)         = Nothing
up    (Loc p x (Push p' c s)) = return (Loc p' (HIn $ fill p c x) s)
right (Loc p x Empty)         = Nothing
right (Loc p x (Push p' c s)) = next (λp z c' → Loc p z (Push p' c' s)) p c x
left  (Loc p x Empty)         = Nothing
left  (Loc p x (Push p' c s)) = prev (λp z c' → Loc p z (Push p' c' s)) p c x

df :: (a → Maybe a) → (a → Maybe a) → (a → Maybe a) → a → Maybe a
df d u lr l =
    case d l of
      Nothing → df' l
      r → r
  where
    df' l =
      case lr l of
        Nothing →
          case u l of
            Nothing → Nothing
            Just l' → df' l'
        r → r

dfnext :: Nav
dfnext = df down up right

dfprev :: Nav
```

$dfprev = df\ down'\ up\ left$

$leave :: Zipper\ \phi\ f \Rightarrow Loc\ \phi\ f\ (HFix\ f)\ ix \rightarrow HFix\ f\ ix$
$leave\ (Loc\ p\ x\ Empty) = x$
$leave\ loc = leave\ (fromJust\ (up\ loc))$

Bibliography

[Bar01] Chris Barker. Iota and jot: the simplest languages? `http://barker.linguistics.fas.nyu.edu/Stuff/Iota/`, 2001.

[BR06] Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In John H. Reppy and Julia L Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16–21, 2006*, pages 216–226. ACM Press, 2006.

[CKPJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005*, pages 1–13. ACM Press, 2005.

[Cor06] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[GEF] The eclipse graphical editing framework (gef). `http://www.eclipse.org/gef/`.

[Hee05] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[Hue97] Gerard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[JDT] The eclipse java development tools. `http://www.eclipse.org/jdt/core/`.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic programming. In *2nd Int. School on Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:168–177, 2009.

[Lin07] Adam Lindsay. LOLCODE. `http://lolcode.com/`, May 2007.

[LM01] 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.

[LMB92] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992.

[LP05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

[McB01] Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract). `http://strictlypositive.org/diff.pdf`, 2001.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, 1991.

[MM01] David Morgan-Mar. Piet. `http://www.dangermouse.net/esoteric/piet.html`, 2001.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[MR07] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell'07*. ACM, 2007.

[Mül93] Urban Müller. Brainfuck. `http://esolangs.org/wiki/Brainfuck`, 1993.

[NRH$^+$08] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 13–24, New York, NY, USA, 2008. ACM.

[OHL06] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, pages 199–216, 2006.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[RHLJ09] Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP 2009*, pages 233–244. ACM, 2009.

[RJJ$^+$08] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell 2008*, pages 111–122. ACM, 2008.

[SAA99] Doaitse Swierstra and Pablo Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. *SOFSEM*, 99:111–129, 1999.

[Sch04] Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, Oct 2004.

[SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[Swi08] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[Vis03] Eelco Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in strategoxt 0.9, 2003.

[VS96] Sreeni Viswanadha and Sriram Sankar. JavaCC. `https://javacc.dev.java.net/`, 1996.

[vS08] Martijn van Steenbergen. Building a hybrid editor. `http://martijn.van.steenbergen.nl/projects/HybridEditor.pdf`, 2008.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Proceedings conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Wad90] Philip Wadler. Recursive types for free! `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`, 1990.