

# Zipper-based Modular and Deforested Computations

Pedro Martins<sup>1</sup>, João P. Fernandes<sup>1,2</sup>, and João Saraiva<sup>1</sup>

<sup>1</sup> High-Assurance Software Laboratory (HASLAB/INESC TEC),  
Universidade do Minho, Portugal

<sup>2</sup> Reliable and Secure Computation Group ((rel)ease),  
Universidade da Beira Interior, Portugal

{prmartins@di.uminho.pt, jpf@di.ubi.pt, jas@di.uminho.pt}

**Abstract.** In this paper we present a methodology to implement multiple traversal algorithms in a functional programming setting. The implementations we obtain are of highly modular and intermediate structure free programs, that rely on the concept of functional zippers to navigate on data structures.

Even though our methodology is developed and presented under *Haskell*, a lazy functional language, we do not make essential use of laziness. This is an essential difference with respect to other attribute grammar embeddings. This also means that an approach similar to ours can be followed in a strict functional setting such as *OCaml*, for example.

In the paper, our technique is applied to a significant number of problems that are well-known to the functional programming community, demonstrating its practical interest.

**Keywords:** Deforested Computation, Generic Programming, Functional Programming

## 1 Introduction

Functional programs are often constructed by gluing together smaller components, using intermediate data structures to convey information between components. These data structures are constructed in one component and later consumed in another one, but never appear in the result of the whole program. This compositional style of programming has many advantages for clarity and modularity, but gives rise to a maintenance problem due to the extra data that must be created and consumed. The usual solution is to remove intermediate data structures by combining smaller components into larger ones, thereby ruining modularity. In this paper we develop a technique for avoiding this tradeoff: we implement modular functional programs without defining intermediate data structures.

Consider the problem of transforming a binary leaf tree  $t_1$  into a new tree  $t_2$  with the exact same shape as  $t_1$ , but with all the leaves containing the minimum value of  $t_1$ . This problem is widely known as *repmim* [1], and is often

used to illustrate important aspects of modern functional languages [2–4]. In this paper, *repm* is also used as a first running example, and we start by presenting different solutions for it. In order to solve *repm*, we start by defining a representation for binary leaf trees:

```
data Tree = Leaf Int | Fork Tree Tree
```

In a strict, purely functional setting, solving this problem requires a two traversal strategy. First, we need to traverse the input tree in order to compute its minimum value:

```
tmin :: Tree -> Int
tmin (Leaf n)      = n
tmin (Fork l r)    = min (tmin l) (tmin r)
```

Having traversed the input tree to compute its minimum value, we need to traverse that tree again. We need to replace all its leaf values by the minimum value:

```
replace :: Tree -> Int -> Tree
replace (Leaf _) m = Leaf m
replace (Fork l r) m = Fork (replace l m)
                          (replace r m)
```

In order to solve *repm*, we now only need to combine functions *tmin* and *replace* appropriately:

```
transform :: Tree -> Tree
transform t = replace t (tmin t)
```

There are many advantages in structuring our programs in this modular way. Considered in isolation, functions *tmin* and *replace* are very clear, simple, they are easy to write and to understand, so they have a great potential for reuse. Furthermore, each function can be focused in performing a single task, rather than attempting to do many things at the same time.

In this particular solution, given the simplicity of *repm*, the input tree *t* serves as input to both functions *tmin* and *replace*. In general, however, modular programs are given by definitions such as *prog* = *f.g*, where *prog* :: *a* → *c*, *g* :: *a* → *b* and *f* :: *b* → *c*. This means that these programs use an intermediate structure, of type *b*, that needs to be more informative than the input one, of type *a*. This fact forces the programmer to define and maintain new data structures which are constructed as the program executes. The construction of these structures that never appear in the result of the whole program adds overhead that makes maintenance hugely difficult, which gets worse as the program increases both in size and in complexity.

In the above solution, it is also the case that the scheduling of computations was left to the programmer. Indeed, in order to implement *transform*, we realized that the minimum of the input tree needs to be computed before the replacement is possible. Although the scheduling in this case is trivial, for more realistic problems, the scheduling of computations may not be a simple task. For example, the optimal pretty printing algorithm presented in [5] is implemented

by four traversal functions, whose scheduling is extremely complex. Moreover, those four functions rely on three (user-defined) gluing intermediate structures to convey information between the different traversals.

In a lazy functional setting such as *Haskell* an alternative solution to *repm* can be formulated. In his original paper, [1] showed how to derive such a solution from the two traversal solution seen before. He derives the program:

```

repm  :: Tree -> Int -> (Tree, Int)
repm  (Leaf n)  m = (Leaf m,  n)
repm  (Fork l r) m = (Fork t1 t2, min m1 m2)
  where (t1, m1) = repm l m
        (t2, m2) = repm r m
transform :: Tree -> Tree
transform t = nt
  where (nt, m) = repm t m

```

This program is *circular*: we can see that, in the definition of the *transform* function, *m* is both an argument and a result of the *repm* call. Although this definition seems to induce both a cycle and non-termination of this program, the fact is that, in a lazy setting, the lazy evaluation machinery is able to determine, at runtime, the right order to evaluate it. In this type of programs, the work associated with the scheduling of computations is, therefore, transferred from the programmer to the lazy evaluation machinery.

We may also notice that the circular version of *transform* does not construct or use any intermediate data structure, and this is a characteristic of all circular programs: since they define a single function to perform all the work (*repm*, in the example), the definition of intermediate data structures to glue different functions loses its purpose. In fact, circular programming may be considered an advanced technique for intermediate structure deforestation [3].

In a circular program, the definition of a single function, on the other hand, forces us to encode together all the variables used in the program. Indeed, if we needed to use more arguments or to produce more results in our example, these would all have to be defined in *repm*. As a consequence, the definition of such a function needs to be concerned with using and computing many different things. In this sense, we observe that circular programs are not modular.

Circular programs are also known to be difficult to write and to understand and even for experienced functional programmers, it is not hard to define a *real* circular program, that is, a program that does not terminate. The execution of such programs is, furthermore, restricted to a lazy execution setting, since such a setting is essential to schedule circular definitions. This means that we are not able to execute the latter version of *transform* in a strict language such as *Ocaml*, for example.

In summary, we notice some characteristics of these approaches: the first version of *transform* is highly modular and its execution is not restricted to a lazy setting, but relies on gluing data types and function scheduling, whereas the second one is free of intermediate structures and requires no explicit scheduling by the programmer but is hard and "non-natural" to write such circular pro-

grams, and even for an advanced lazy functional programmer it is hard to write a program which is not completely circular, i.e., which terminates.

In this paper, we develop a framework for implementing multiple traversal algorithms in a functional setting. Programs in our framework combine the best of the two *transform* solutions: they are modular, intermediate structure free and do not require explicit scheduling by the programmer. This is achieved by thinking of our programs in terms of Attribute Grammars (AGs), i.e., by implementing AGs as first class elements in our language. In this sense, our work may also be thought of as an AG embedding.

In the literature, one may find other approaches with similar goals; [2, 4] are two notable examples. An essential difference with respect to these approaches is that our framework does not make essential use of laziness, so that it can easily be implemented in a strict setting such as *Ocaml*. A more detailed comparison to related work is presented in section 6.

The framework we propose relies heavily on the concept of functional zippers, originally proposed by [6]. As we will see later, our use of functional zippers is such that they provide an elegant and efficient mechanism for navigating on tree structures, but also to hide that navigation.

In a previous work [7], we have presented an embedding of Attribute Grammars in a functional setting, together with modern AG extensions, and shown how these can be used to implement the semantics of programming languages. In this work, we show how such setting can also be useful as an alternative to traditional implementations on a functional setting, by creating programs that are more modular and structured.

*This paper is organized as follows.* In section 2, we review the standard concept of functional zippers. In section 3, we show how standard zippers can be used to express a modular attribute grammar to solve the *repmim* problem. Our approach is then used in sections 4 to 5 to express attribute grammar solutions to programming problems more realistic than *repmim*, as well as a generic solution to *repmim*. In 6 we describe works that relate to ours, and finally in section 7 we draw our conclusions.

## 2 The zipper

The zipper data structure was originally conceived by Huet [6] to solve the problem of representing a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down the tree.

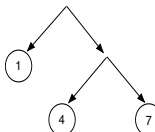
In our work we have used the generic zipper library of [8]. It works for both homogeneous and heterogeneous datatypes, and data-types for which an instance of the *Data* and *Typeable* type classes [9] are available can be traversed.

In order to introduce the concept of zipper, we consider again the representation for binary leaf trees in *Haskell*:

```
data Tree = Leaf Int | Fork Tree Tree
```

and one of its possible instances with its visual representation:

```
tree = Fork (Leaf 1)
          (Fork (Leaf 4)
                (Leaf 7))
```



We may notice that, in particular, each of *tree*'s subtree occupies a certain location in *tree*, if we consider it as a whole. That location may be represented by the subtree under consideration and by the rest of the tree, which is viewed as the context of that subtree. For instance, the context of *Leaf 4* in *tree* is

```
Fork (Leaf 1)
     (Fork focus
         (Leaf 7))
```

where *focus* marks a hole which corresponds exactly to the spot where *Leaf 4* appears in *tree*. One of the possible ways to represent this context is as a path from the top of the tree to the hole. To reach *Leaf 4* in *tree*, we need to go down the right branch and then down the left one.

Using this idea, we can easily reach *Leaf 4* in *tree* using the generic zippers provided by [8]. We start by encapsulating *tree* into a zipper:

```
a = toZipper tree
```

where *a* has the type  $a :: \text{Zipper Tree}$ . With this, it is simple to traverse *a* and the position of the tree where *Leaf 4* is:

```
b = let d = fromJust (down a)
     in fromJust (down' d)
```

In this operation, we go to the rightmost child using *down*, and then to the leftmost child using *down'*. Since all the functions of this library wrap the result inside a data type to make them total, we also have to unwrap the result every time a function is called. We do so simply by using the function *fromJust*<sup>3</sup>.

The result of this operation has the type  $b :: \text{ZipperTree}$ , meaning that *b* is a zipper like *a*, with the difference of having a different *focus*. With this said, if we ask for the focus of *b*, using the function *getHole*:

```
let focus = getHole b :: Maybe Tree
focus = Leaf 4
```

In this formalism, the semantics are dependent on information that is immediately above or below of a certain tree position, concept which is directly provided by zippers and the associated navigation functions.

The zipper data structure provides an elegant and efficient way of manipulating locations inside a data structure. Zippers are particularly useful for performing incremental edits on tree structures. Zippers have already been used in the implementation of filesystems [10] and window managers [11], but they are

<sup>3</sup> We are not really checking for totality, otherwise we would have to test each function call against a set of possible results. For simplicity we are just assuming the function produced a result and we are directly unwrapping it with *fromJust*.

applicable anytime there is a focal point for edits. In the implementation of a filesystem, the current working directory is the focal point of attention, and in a window manager it is the window with focus.

In the next section, we show how we abstract this library of zippers by creating a set of constructors that resemble more closely the traditional formalism to implement AGs.

## 2.1 Abstracting the Generic Zippers

The generic zipper library presented on the previous section provides a useful set of functions to navigate throughout data types. However, it is our intention to abstract as much as possible from this library and create a setting where *Haskell* constructors are as similar as possible to the typical primitives used in Attribute Grammars. In this section we present a set of functions that allows the easy navigation of data types, that does not require further testings for the user (for example, we abstract over totality checks) and leverages the implementation to one much closer to AGs.

Let us consider a concrete data type to represent programs in an Algol 68-like language restricted to expressing declarations and uses of variables. Programs in this language consist of instruction blocks, where each instruction declares a variable, uses a variable or defines a nested instruction block. A small example of a program in this language is

```
p = [ decl ' y; [ decl ' w; use ' x; ] use ' y; ]
```

In order to represent programs in the Algol 68 language, we define the following *Haskell* data-type. This data-type will be used, in Section 4, to implement a semantic analyzer for that language:

```
data Root = Root Its
        deriving (Typeable, Data)
data Its = ConsIts It Its | NilIts
        deriving (Typeable, Data)
data It = Decl String | Use String | Block Its
        deriving (Typeable, Data)
```

In this representation,  $p$  is defined as:

```
p = Root (ConsIts (Decl "y")
             (ConsIts (Block (ConsIts (Decl "w")
                                     (ConsIts (Use "x") NilIts)))
                     (ConsIts (Use "y") NilIts)))
```

Our goal now is to navigate on elements of type  $P$  in the same way that we traversed elements of type  $Tree$ , in Section 2. Using our Attribute Grammar-based approach, instead of writing concrete location navigation functions, the user is only required to declare the data types to traverse as deriving from the *Data* and *Typeable* type classes, which are provided as part of GHC's<sup>4</sup> libraries.

<sup>4</sup> The Glasgow Haskell Compiler, <http://www.haskell.org/ghc/>

This means we will be immediately able to navigate through the data types *Root*, *Its* and *It* using the zipper-provided functions.

Suppose that we want to traverse the previous *Algol* phrase to the identifier of the variable that is being used in the nested block of *p*. The first thing to do is to get *p* inside a zipper:

```
g1 :: Zipper Root
g1 = toZipper (Root p)
```

In order to reach the desired nested block on the program *p*, we now need to go down from the root location created in *g1*. We do this using function *.\$*, as follows:

```
g2 = g1.$1 :: Zipper Root
```

Data type locations do not need any information about the types of their children, but neither does the user. Because we are embedding *Algol* in *Haskell*, and *Haskell* has a strong type system, type correctness is always necessarily enforced meaning a phrase of *Algol* is always type-valid. And because the zipper does not need any contextual information, regarding what is above or below a given position, this information is abstracted from the user as well. Of course, one must be aware of the position of a tree where certain computation needs to be performed, but our setting adds an increased level of abstraction comparing with traditional *Haskell* programs while retaining the same core language features such as type safety or referential transparency.

Returning to our example, the value held by the location *g2* is one position below the initial focus on the zipper, which was *Root*. As expected, *g2* yields:

```
g2 = (ConsIts (Decl "y")
      (ConsIts (Block (ConsIts (Decl "w")
                              (ConsIts (Use "x") NilIts)))
              (ConsIts (Use "y") NilIts))))
```

We need to continue going down on *p*, if we want to edit the declaration of the variable *w*:

```
g1 = g2.$2
```

With *g3*:

```
g1 = ConsIts (Block (ConsIts (Decl "w")
                              (ConsIts (Use "x") NilIts)))
            (ConsIts (Use "y") NilIts))
```

Notice that function *.\$* is a generic function, that applies to locations on any data type that derives from *Typeable* and *Data*. It is not even the case that *.\$* applies only to data-types that have the same number of children. What is more, *Root* has a single data-type child, *Its*, and that *Its* may have two children, *It* and *Its*. When *.\$* is applied with a constructor that has more than one child, it will go down to the user-defined one. This is precisely how the original Attribute Grammars formalism works: semantics on a tree site depends on the parent if

they are inherited, or on specific children that in our setting are numerically defined.

We can see that the declaration of variable "w" occurs in the first (left) child of the current location. This means that if we apply function `.$1` to  $g_3$  we will immediately go to the correct child:

```
g4 = g1 . $1
```

Location  $g_4$  holds, as expected, the nested block of instructions that occurs both in  $p$ , and in  $g_3$ :

```
g4 = Block ( ConsIts (Decl "w")
              ( ConsIts (Use "x") NilIts ))
```

We continue our navigation performing another *go down* step to access the instructions in the nested block on  $g_4$ ,

```
g4 = g4 . $1
```

obtaining:

```
g4 = ConsIts (Decl "w")
           ( ConsIts (Use "x") NilIts )
```

An important remark is that in this block ( $g_5$ ) we declare a variable "w" but use a variable "x"; intuitively, the identifiers of these two variables should probably match. The zipper library we use provides primitives to change the parts of a tree. In this example, we could easily correct the wrong assignment of "x" (or declaration of "w"). We do not worry about this as this is not a traditional behavior of Attribute Grammars.

Attribute Grammars as a formalism is extremely suitable to perform tree transformations, but such operations are typically implemented by designing a set of attributes that traverses the tree and whose result is a new, refactored one. With this in mind, using AGs we would not change our zipper, we would instead create a new, corrected tree (as we do in Sections 3 and 5). Nevertheless, such operation is possible and the functions provided in the generic zipper library we use are compatible with our abstraction of tree navigation functions with `($)`, serving as a reminder of the adaptability of our approach.

In the next sections, we show how the generic zipper framework introduced in this section can be used to solve different programming problems.

### 3 The repmin

In [1], it was originally proposed to solve *repmin* using a circular program, i.e., a program where, in the same function call, one of its results is at the same time one of its arguments. With his work, Bird showed that any algorithm that performs multiple traversals over the same data structure can be expressed in a lazy language as a single traversal circular program.

Furthermore, using circular programming, the programmer does not have to concern himself with the definition and the scheduling of the different traversal functions and, because there is a single traversal functions, neither does the



programmer have to define intermediate gluing data structures to convey information between traversals.

Writing circular programs, however, forces the programmer to encode together all the arguments and results that are used in the circular call. When functions have many arguments as well as many results, it is often preferable to express multiple traversal algorithms in terms of attribute grammars (AGs), that have been proved to be strongly related to circular programs [12, 13]. The AG programming paradigm does not force the programmer to encode all the aspects together.

Returning to our example, in [4], the authors identified three components for solving *repm*: computing the minimal value, passing down the minimal value from the root to the leaves and constructing the resulting tree. In this section, we review the Attribute Grammar for *repm* that was introduced by [14], and show how each of the three components identified by [4] in that grammar can be embedded in *Haskell* using our approach.

The attribute grammar for *repm* starts by defining the underlying data structure, i.e., binary leaf trees. The attribute grammar fragments presented in this section follow the standard AG notation of [14]. In this notation, we straightforwardly use the *Tree* datatype from Section 2.

Having defined the structure, we need to define functionality. We start by reviewing the AG component that computes the minimal value of a tree:

```

SYN Tree [smin : Int]
SEM Tree | Leaf lhs.smin = @v
          | Fork lhs.smin = @left.smin
                           'min'
                           @right.smin

```

This component declares, using the *SYN* keyword, that elements of type *Tree* synthesize an attribute *smin* of type *Int*. Then, a *SEM* sentence defines how *smin* is computed: when the current tree is a leaf, clearly its minimal value is the leaf value itself; when it is the fork of two other trees (the *left* and the *right* subtrees), we compute the minimal values of each subtree (i.e., their *smin* attribute), and then their minimal value (function *min*). In this notation, *lhs* refers to the left-hand side symbol of the production and @ prefixes a reference to a field.

Our zipper-based embedding of this component is defined as:

```

smin :: Zipper Root -> Int
smin t = case constructor t of
  "Root" -> smin (t.$1)
  "Leaf" -> lexeme t
  "Fork" -> min (smin (t.$1)) (smin (t.$2))

```

Function *constructor*, given in Section 3.1, maps any element to a textual representation of its constructor. Function *lexeme* is also defined in Section 3.1 to compute the concrete value in any leaf of a tree.

We can see that the embedding of *smin* that we obtain very much directly follows from its AG specification.

Having implemented the first of the three components that solve *repm*, we now consider the remaining two. We start by implementing the construction of the result of *repm*, a tree with all leaves being the minimum of the original one.

```

SYN Tree [sres : Tree]
SEM Tree | Leaf lhs.sres = Leaf @lhs.ival
        | Fork lhs.sres = Fork @left.sres @right.sres

```

We are now defining an attribute *sres*, again synthesized by elements of type *Tree*. This attribute definition may again be mapped to our setting very easily. We obtain the following implementation:

```

sres :: Zipper Root -> Tree
sres t = case constructor t of
  "Root" -> sres ( t.$1 )
  "Leaf" -> Leaf (ival t)
  "Fork" -> Fork (sres (t.$1)) (sres (t.$2))

```

The implementation of *sres* places in each leaf of a tree the value of the *ival* attribute. This value corresponds to the minimal value of the global tree, that still needs to be passed down to all the nodes in the tree. This corresponds exactly to the third component that we still need to implement. In order to bind the minimal value being computed (attribute *smin*) with the minimal value that is passed down through the tree (attribute *ival*), it is common, in the AG setting, to add a new data-type definition,

```
DATA Root | Root tree : Tree
```

and a new semantic rule,

```
SEM Root | Root tree.ival = @tree.smin
```

Passing down *ival* then becomes:

```

INH Tree [ival : Int]
SEM Tree | Fork left .ival = @lhs.ival
        | Fork right .ival = @lhs.ival

```

In our setting, we closely follow this approach and always introduce a new data type that marks the topmost position of the tree, in this case, "*Root*".

When the current location corresponds to the top one, we have to define the values of *smin* to *ival* in this particular position. In our setting, we then define the attribute *ival* as follows (we use the location navigation function *parent* to access the parent of a tree location):

```

ival :: Zipper Root -> Int
ival t = case constructor t of
  "Root" -> smin t
  "Leaf" -> ival (parent t)
  "Fork" -> ival (parent t)

```

Notice that we do not explicitly distinguish between inherited and synthesized attributes. Like in modern attribute grammar systems [14, 15], inherited attributes, such as *ival*, correspond to attributes that are defined in parent nodes.

Having defined the three components that allow us to solve the *repm* problem, we may now define a semantic function that takes a tree and produces a *repm* tree, using these components:

```
semantics :: Root -> Tree
semantics t = replace (toZipper t)
```

Regarding this implementation, we may notice that it has the best properties of the two *transform* solutions presented in section 1. First, it is modular, since the global computational effort has been separated into several components, and does not rely on laziness, like the first *transform* implementation. Second, it constructs no intermediate structure and it requires no explicit scheduling; notice that *sm*, *replace* and *ival* were defined with no particular focus on the order they need to be computed.

In this section, we have presented a solution to the *repm* problem in terms of an attribute grammar. Our solution is expressed in *Haskell* and closely follows common attribute grammar notation. We showed that we can easily intermingle separate concerns with the implementation's basic functionality, which itself has been split into different components. Therefore, we believe that our framework is very appropriate for static aspect oriented programming [16] in a functional language.

### 3.1 Boilerplate Code

Our goal with this paper was to address concerns of the expression problem without relying in an opaque and possibly complex pre-processor. A disadvantage of this approach is that some code that may be considered boilerplate code needs to be manually defined. Function *constructor*, that we have used throughout the paper to map an element to a textual representation of its constructor, is a function clearly in this set: it goes down all the possible constructors and tries to match them with a given element.

```
constructor :: Zipper Root -> String
constructor a = case (getHole a :: Maybe Tree) of
  Just (Fork _ _) -> "Fork"
  Just (Leaf _)   -> "Leaf"
  otherwise -> case (getHole a :: Maybe Root) of
    Just (Root _) -> "Root"
```

For each individual argument, *constructor* matches an element wrapped up in a zipper against the constructors of the data-type *Tree*. In this example, *constructor* matches an element against the constructors *Fork*, *Leaf* and *Root*, creating a *String* representation of them.

The other function that we have used and that could easily be given by a pre-processor is function *lexeme*, that computes the value in any leaf of a tree.

In the leaves of our running example's trees, we only have elements of one data constructor, *Leaf*, and these elements are always of type *Int*. So, it suffices to define:

```
lexeme :: Zipper Root -> Int
lexeme t = let Leaf v = fromJust (getHole t :: Maybe Tree)
           in v
```

As we said, the functions defined in this section could easily be given by a pre-processor. Indeed, we could have implemented a simple program to go through the Haskell's abstract syntax tree that we obtain by parsing the data-types definitions of Section 3 and finding all its constructors (for *constructor*) and all its leaves (for *lexeme*). We, however, opted not to use this pre-processor approach because we wanted to give clear and transparent definitions for all the functions involved in our framework that also could be edited in a simple way by any programmer. What's more, the use of a pre-processor is often considered a disadvantage when one needs to choose to re-use a tool or a setting against some others.

## 4 The Algol 68 scope rules

In this section we present an attribute grammar that uses the tree navigating mechanism of the generic zipper presented in the previous sections to implement the Algol 68 scope rules [17]. These rules are used, for example, in the Eli system [18] to define a generic component for the name analysis task of a compiler.

We wish to construct a modular and deforested program to deal with the scope rules of the block structured language introduced in Section 2.1. If the reader recalls the running example, named "*p*", there was an identifier "*x*" which was visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of "*x*". In the latter case, the definition of "*x*" in the local scope hides the definition in the global one. In a block an identifier may be declared at most once. In Section 2.1, *p* is a simple example of a program we want to analyze. The following program illustrates a more complex situation where an inner declaration of "*y*" hides an outer one.

```
p' = [ use ' y; decl ' x;
      [ decl ' y; use ' y; use ' w; ]
      decl ' x; decl ' y; ]
```

Programs such as *p* or *p'* describe the basic block-structure found in many languages, with the peculiarity that declarations of identifiers may also occur after their first use. According to these rules, *p'* contains two errors: a) at the outer level, the variable "*x*" has been declared twice, and b) the use of the variable "*w*", at the inner level, has no binding occurrence at all.

We aim to develop a program that analyses *Algol* programs and computes a list containing the identifiers which do not obey the scope rules. In order to make it easier to detect which identifiers are being incorrectly used in a program, we

require that the list of invalid identifiers follows the sequential structure of the program. Thus, the semantic meaning of processing  $p'$  is  $[w, x]$ .

Because we allow use before declaration, a conventional implementation of the required analysis leads to a program which traverses the abstract syntax tree twice: once to accumulate the declarations of identifiers and construct an environment, and again to check the uses of identifiers using the computed environment. The uniqueness of names is detected in the first traversal: for each newly encountered declaration we check whether the identifier has already been declared at the current level. In this case an error message is computed. Of course the identifier might have been declared at an outer level. Thus we need to distinguish between identifiers declared at different levels. We use the level of a block to achieve this. The environment is a partial function mapping an identifier to its level of declaration.

As a consequence, semantic errors resulting from duplicate definitions are computed during the first traversal of a block and errors resulting from missing declarations in the second one. A straightforward implementation of this program may be sketched as<sup>5</sup>:

```

semantics :: P -> Errors
semantics p = missing_decls (duplicate_decls p)
duplicate_decls :: P -> (P', Env)
missing_decls   :: (P', Env) -> Errors

```

In this implementation, a "gluing" data structure, of type  $P'$ , has to be defined by the programmer and is constructed to pass the detected errors explicitly from the first to the second traversal, in order to compute the final list of errors in the desired order. To be able to compute the missing declarations of a block, the implementation also has to explicitly pass the names of the variables that are used in a block between the two traversals of the block. This information must therefore also be in the  $P'$  intermediate structure.

We start by defining an *Haskell* datatype that describes *Algol* syntactically, whose data constructors will be, similarly to AGs, used as semantic points on which functions (read attributes) will be defined.

```

data Root = Root Its

data Its = ConsIts It Its
         | NilIts

data It = Decl String
         | Use String
         | Block Its

```

Next, we implement the same analysis but in terms of an attribute grammar that does not rely on the construction of any intermediate structure.

As stated before, the language presented in this chapter does not force a *declare – before – use* discipline, which means a conventional implementation

<sup>5</sup> The interested reader may find in [17, 19] strict and circular solutions to solve these scope rules.

of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and a second time to process the uses of identifiers (using the computed environment) in order to check for the use of non-declared identifiers.

An algorithm for processing this language as to be designed in two traversals:

- On a first traversal, the algorithm has to collect the list of local definitions and, secondly, detect duplicate definitions from the collected ones
- On a second traversal, the algorithm has to use the list of definitions from the previous step as the global environment, detect the use of non-defined variables and finally combine the errors from both traversals.

Next, we will define the semantics of the grammar. For every block we compute three things: its environment, its lexical level and its invalid identifiers. The environment defines the context where the block occurs. It consists of all the identifiers that are visible in the block. The lexical level indicates the nesting level of a block. Observe that we have to distinguish between the same identifier declared at different levels, which is a valid declaration (for example, "decl y" in  $p'$ ), and the same identifier declared at the same level, which is an invalid declaration (for example, "decl x" in  $p'$ ). Finally, we have to compute the list of identifiers that are incorrectly used, i.e., the list of errors.

The Attribute Grammar that analyses a phrase of *Algol* will be composed by:

- An environment, attribute *env*, which consists of all the identifiers that are visible in the block:  $type\ Env = [(String, Int)]$
- A lexical level, attribute *lev*, which indicates the nesting level of a block:  $type\ Level = Int$
- The invalid identifiers, attribute *errs*, which contains the list of identifiers that are incorrectly used:  $type\ Errors = [String]$

We start by defining the construction of the environment of an *Algol* program. Every block inherits the environment of its outer block. Therefore, we associate an inherited attribute *dcli*, that carries an environment, to the non-terminal symbols *Its* and *It* that define a block. The inherited environment is threaded through the block in order to accumulate the local definitions and in this way synthesizes the total environment of the block. We associate a synthesized attribute *dclo*, that also carries the environment, to the non-terminal symbols *Its* and *It*, which defines the newly computed environment.

In our solution, we defined semantic *Haskell* functions which pattern match on data constructors. For the readers familiar with Attribute Grammars, there is an obvious mapping between *Haskell* functions and attributes, and between data constructors and grammar productions. The attributes *dcli* and *dclo* are declared as follows:

```
dcli :: Zipper Root -> [(String, Int)]
dcli z = case (constructor z) of
```

```

"Root" -> []
"NilIts" -> case (constructor (parent z)) of
  "ConsIts" -> dclo ((parent z).$1)
  "Block" -> env (parent z)
  "Root" -> []
"ConsIts" -> case (constructor (parent z)) of
  "ConsIts" -> dclo ((parent z).$1)
  "Block" -> env (parent z)
  "Root" -> []
"Block" -> dcli (parent z)
"Use" -> dcli (parent z)
"Decl" -> dcli (parent z)

```

```

dclo :: Zipper Root -> [(String, Int)]
dclo z = case (constructor z) of
  "ConsIts" -> dclo (z.$2)
  "NilIts" -> dcli z
  "Use" -> dcli z
  "Decl" -> (value z, lev z) : (dcli z)
  "Block" -> dcli z

```

The only production that contributes to the synthesized environment of a phrase of *Algol* is *Decl*. The single semantic equation of this production makes use of the semantic function `'` (written in infix notation) to build the environment. Note that we are using the *Haskell* type definition presented previously. The use of pairs is used to bind an identifier to its lexical level. The single occurrence of pseudo-terminal *Name* is syntactically referenced in the equation since it is used as a normal value of the semantic function. All the other semantic equations of this fragment simply pass the environment to the left-hand side and right-hand side symbols within the respective productions.

Now that the total environment of a block is defined, we pass that context down to the body of the block in order to detect applied occurrences of undefined identifiers. Thus, we define a second inherited that also carries the environment, called *env*, to distribute the total environment. It should be noticed that attribute *dclo* can be used to correctly compute the required list of errors. We choose to distribute the list of declarations in a new attribute to demonstrate our techniques, as with this approach we force a two traversal (strict) evaluation scheme. Although this approach is not really needed in the trivial *Algol* language, it is a common feature when defining real languages. *Env* is defined as:

```

env :: Zipper Root -> [(String, Int)]
env z = case (constructor z) of
  "NilIts" -> case (constructor (parent z)) of
    "Block" -> dclo z
    "ConsIts" -> env (parent z)
    "Block" -> dclo z
  "ConsIts" -> case (constructor (parent z)) of
    "Block" -> dclo z

```

```

"ConsIts" -> env (parent z)
"Root"    -> dclo z
"Block"   -> env (parent z)
"Use"     -> env (parent z)
"Decl"    -> env (parent z)
"Root"    -> dclo z

```

The first semantic equation of *Block* specifies that the inner blocks inherit the environment of their outer ones. As a result, only after computing the environment of a block is it possible to process its nested blocks. That is, inner blocks will be processed in the second traversal of the outer one.

The total environment of the inner blocks, however, is the synthesized environment (attribute *dclo*), as defined for *Block*. It is also worthwhile to note that the equations:

```

"Root" -> dclo z
"Block" -> dclo z

```

induce a dependency from a synthesized to an inherited attribute of the same symbol.

Every block has a lexical level. Thus, we introduce one inherited attribute *lev* indicating the nesting level of a block. The *Haskell* primitive function '+' is used to increment the value of the lexical level passed to the inner blocks:

```

lev :: Zipper Root -> Int
lev z = case (constructor z) of
  "Root"    -> 0
  "NilIts"  -> case (constructor $ parent z) of
    "Block"   -> (lev (parent z)) + 1
    "ConsIts" -> lev (parent z)
    "Root"    -> 0
  "ConsIts" -> case (constructor (parent z)) of
    "Block"   -> (lev (parent z)) + 1
    "ConsIts" -> lev (parent z)
    "Root"    -> 0
  "Block"   -> lev (parent z)
  "Use"     -> lev (parent z)
  "Decl"    -> lev (parent z)

```

Finally, we have to synthesize one attribute defining the (static) semantic errors. We define a second synthesized attribute: *errs*. The attribution rules for this semantic domain are shown next:

```

errs :: Zipper Root -> [String]
errs z = case (constructor z) of
  "Root"    -> errs (z.$1)
  "NilIts"  -> []
  "ConsIts" -> (errs (z.$1)) ++ (errs (z.$2))
  "Use"     -> mBIn (value z) (env z)
  "Decl"    -> mNBIn (value z, lev z) (dclo z)
  "Block"   -> errs (z.$1)

```



There are two semantic functions that we need to define:  $mBIn$  and  $mNBIn$ . The definition of these functions must be included in the grammar specification. For this reason, attribute grammar specification languages provide an additional notation in which semantic functions can be defined. Generally, this notation is simply a standard programming language. We are embedding AG's so we use plain *Haskell* for these functions. Thus, the two semantic functions look as follows:

```

mBIn :: String -> [(String, Int)] -> [String]
mBIn name [] = [name]
mBIn name ((n,l):es) = if (n==name) then []
                        else mBIn name es

mNBIn :: (String, Int) -> [(String, Int)] -> [String]
mNBIn tuple [] = []
mNBIn pair (pl:es) = if (pair==pl) then [fst pair]
                        else mNBIn pair es

```

We may now define a program that implements the semantic analysis described, simply by inspecting the *errs* attribute computed at the topmost location of the program:

```

semantics :: P -> [String]
semantics p = errs (toZipper p)

```

This program can be used to compute the list of errors occurring in the  $p$  and  $p'$  programs presented before. As expected, we obtain:

```

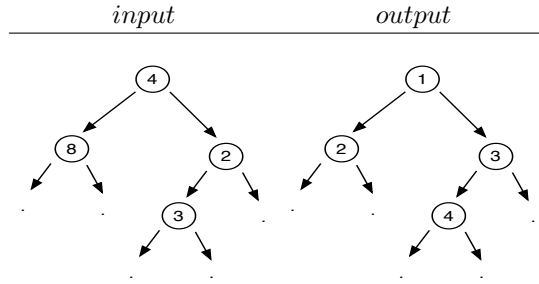
semantics p = ["x"]
semantics p' = ["w", "x"]

```

## 5 Breadth-first numbering

The running examples presented so far have shown that zippers provide a modular and intermediate structure free environment for implementing multiple traversal algorithms in a functional setting. A key aspect of the implementations seen earlier in the paper is that they make no essential use of laziness. In fact, all of these implementations could be *straightforwardly* translated and implemented in a strict setting. This property does not hold for the example that we study in this section.

Consider the problem, described in detail in [20], of breadth first numbering a binary tree, or *bfN* for short. A sample input/output to such problem is sketched next.



In order to tackle this problem, we follow the approach taken by [20]. To implement *bfm*, the author computes a list of integers representing the first available index on each of the levels of the input tree. This list is initially the infinite list of ones and is updated as it goes down the tree to produce the numbering.

In order to implement this algorithm we will need three attributes. Attribute *slist* will be used to compute the list of indexes and *ilist* to pass that list down the tree. Attribute *replace* will hold the result of breadth-first numbering a tree. Attributes *slist* and *replace* have the same definition for any tree, regardless of whether it is the topmost one or one of its subtrees. In this example, we follow Okasaki by using binary trees instead of binary leaf trees:

```
data Tree = Fork Int Tree Tree | Empty
      deriving (Typeable, Data)
```

Attributes *slist* and *replace* are defined as follows, for any *tree* location:

```
slist :: Zipper Root -> [Int]
slist z = case (constructor z) of
  "Fork"  -> (head (ilist z) + 1) : (slist (z.$3))
  "Empty" -> ilist z

replace :: Zipper Root -> Tree
replace z = case (constructor z) of
  "Empty" -> Empty
  "Fork"  -> Fork (head (ilist z))
              (replace (z.$2)) (replace (z.$3))
  "Root"  -> replace (z.$1)
```

The third attribute, *ilist*, is a little bit more tricky. We have to defined *ilist* for the upmost location on the input tree, which we do by testing if the parent is the "Root". We then define the following values for *ilist*:

```
ilist :: Zipper Root -> [Int]
ilist z = case (constructor (parent z)) of
  "Root" -> [1] ++ (slist z)
  {- If z is the third child, it is the rightmost one-}
  otherwise -> case (z.|3) of
    True -> slist (fromJust (left z))
    False -> tail (ilist (parent z))
```

Notice the very peculiar relationship between attributes *ilist* and *slist* at the top level: *ilist* is defined as the list whose head is 1 and whose tail is *slist*, and *slist* is defined as the list whose head is the head of *ilist* incremented by 1 and whose tail is the *slist* value computed for the right subtree of the current tree. Then, if we try to compute, for example, the value of *ilist* in a strict setting, this will cause the value of *slist* to be fully computed. But *slist* can not be computed until *ilist* is itself computed. Therefore, in a strict setting, these computations can not be ordered, and this particular program can not be directly implemented in such a setting. In a lazy setting, however, the use of *head*, the standard operator that selects the first element of a list, makes it possible for the above program to terminate. So, even though our approach does not fundamentally depend on laziness, attribute definitions that use laziness can be accommodated.

It is now simple to obtain a *bfn* transformer for binary trees:

```
transform :: Tree -> Tree
transform t = replace (toZipper (Root t))
```

## 6 Related Work

In this paper, we have shown how the zipper data structure can be used to implement multiple traversal algorithms in a functional language. The implementations we obtain are modular, do not require the use of intermediate data structures and do not fundamentally rely on laziness. That is to say that our implementations benefit from the best of the two traditional ways of expressing multiple traversal programs described in the introduction.

Ustalu and Vene [21] use zippers in their approach to embed computations using comonadic structures, with tree nodes paired with attribute values. However, the zipper approach they use does not appear to be generic and must be individually instantiated for each new structure. They also rely on laziness to avoid static scheduling.

Zippers are also used by Badouel *et al.* [22], where zipper transformers define evaluations. This approach relies on laziness and their zipper representation is not generic. This is also the case of [23], that similarly requires laziness and forces the programmer to be aware of a cyclic representation of zippers.

Yakushev *et al.* [24] use mutually recursive data types, for which operations are described with a fixed point strategy. In this work, data structures are translated into generic representations, used for traversals and updates, and translated back after. This solution implies the extra overhead of the translations, and also requires advanced features of *Haskell* such as type families and rank-2 types.

We have showed previously how zippers can be used to embed AGs on a functional setting, together with modern extensions [7, 25]. Even though our library is defined in *Haskell*, a lazy language, we do not make essential use of laziness, making the approach extendable to strict languages.

With this work we further extend the functionalities of functional zippers and show these can be used as a substitute to traditional programming techniques

in a functional setting: while we do not rely on laziness, we present a setting where the programmer can abstract from function scheduling and intermediate data types and focus on more modular programs.

## 7 Conclusions

In this paper we presented a zipper-based approach to elegantly and modularly express circular programs in a functional setting. Our approach does not rely on laziness such as circular programs do, and does not force the programmer to deal with intermediate data structures nor to schedule multiple traversal functions. Our solution uses functional zippers as a mechanism to allow generic tree traversals upon which traversal functions are defined.

We have further proof-tested our approach by embedding other languages in *Haskell*, using implementations that avoid functions scheduling and intermediate data structures. These, together with the examples from this paper, can be found in [www.di.uminho.pt/~prmartins](http://www.di.uminho.pt/~prmartins) or in the cabal package *zipperAG*.

As future work we plan to study both the design and implementation of our embedding when compared to other techniques. Thus, we plan to study how our embedding compares to first class AGs [2, 26]. Circular programs are known to have some performance overhead due to lazy evaluation. We want to study the performance of the zipper embedding, and how the strictification techniques presented in [27] could be adapted to our setting.

## References

1. Bird, R.: Using circular programs to eliminate multiple traversals of data. *Acta Informatica* **21** (1984) 239–250
2. de Moor, O., Backhouse, K., Swierstra, S.D.: First-class attribute grammars. *Informatica (Slovenia)* **24**(3) (2000)
3. Fernandes, J.P., Pardo, A., Saraiva, J.: A shortcut fusion rule for circular program calculation. In: *Proceedings of the ACM SIGPLAN Haskell Workshop*. (2007) 95–106
4. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: *Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'09)*. (2009) 245–256
5. Swierstra, D., Chitil, O.: Linear, bounded, functional pretty-printing. *Journal of Functional Programming* **19**(01) (January 2009) 1–16
6. Huet, G.: The zipper. *Journal of Functional Programming* **7**(5) (1997) 549–554
7. Martins, P., Fernandes, J., Saraiva, J.: Zipper-based attribute grammars and their extensions. In Bois, A.R., Trinder, P., eds.: *Programming Languages*. Volume 8129 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg (2013) 135–149
8. Adams, M.D.: Scrap your zippers: a generic zipper for heterogeneous types. In: *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming. WGP '10*, New York, NY, USA, ACM (2010) 13–24
9. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: *Proc. of the 2003 ACM SIGPLAN Inter. Workshop on Types in Language Design and Implementation. (TLDI '03)*, ACM (2003) 26–37

10. Kiselyov, O.: Tool demonstration: A zipper based file/operating system. In: Haskell Workshop. ACM Press (September 2005)
11. Stewart, D., Janssen, S.: XMonad: A tiling window manager. In: Haskell '07: Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell, ACM Press (2007)
12. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Functional Programming Languages and Computer Architecture. (1987) 154–173
13. Kuiper, M., Swierstra, D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands CSN'87. (November 1987)
14. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In Swierstra, D., Henriques, P., Oliveira, J., eds.: Third Summer School on Advanced Functional Programming. Volume 1608 of LNCS Tutorial., Springer-Verlag (September 1999) 150–206
15. Swierstra, D., Baars, A., Löh, A.: The UU-AG attribute grammar system (2004)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
17. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands (December 1999)
18. Kastens, U., Pfahler, P., Jung, M.T.: The eli system. In: CC '98: Proceedings of the 7th Int. Conf. on Compiler Construction, London, UK, Springer-Verlag (1998) 294–297
19. Fernandes, J.P.: Design, Implementation and Calculation of Circular Programs. PhD thesis, Department of Informatics, University of Minho, Portugal (March 2009)
20. Okasaki, C.: Breadth-first numbering: lessons from a small exercise in algorithm design. ACM SIGPLAN Notices **35**(9) (2000) 131–136
21. Uustalu, T., Vene, V.: Comonadic functional attribute evaluation. Trends in Functional Programming, Intellect Books (10) (2005) 145–162
22. Badouel, E., Fotsing, B., Tchougong, R.: Yet another implementation of attribute evaluation. Research Report RR-6315, INRIA (2007)
23. Badouel, E., Fotsing, B., Tchougong, R.: Attribute grammars as recursion schemes over cyclic representations of zippers. Electronic Notes Theory Computer Science **229**(5) (2011) 39–56
24. Yakushev, A.R., Holdermans, S., Löh, A., Jeurig, J.: Generic programming with fixed points for mutually recursive datatypes. In: Procs. of the 14th ACM SIGPLAN International Conference on Functional programming. (2009) 233–244
25. Martins, P.: Embedding Attribute Grammars and their Extensions using Functional Zippers. PhD thesis, Universidade do Minho (2014)
26. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in haskell. SIGPLAN Not. **44**(9) (August 2009) 245–256
27. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. PEPM '11, New York, NY, USA, ACM (2011) 131–140