Modern type-safe embedding of attribute grammars

An exercise with functional zippers in Haskell

João Paulo Fernandes[1], Pedro Martins[2], Alberto Pardo[3], João Saraiva[4], Marcos Viera[3], and Tom Westerhout[5]

[1] CISUC – Universidade de Coimbra, Portugal jpf@dei.uc.pt
[2] University of California, Irvine, USA pribeiro@uci.edu
[3] Universidad de la República, Uruguay, {pardo,mviera}@fing.edu.uy
[4] Universidade do Minho, Portugal, saraiva@di.uminho.pt
[5] Radboud University, The Netherlands, t.westerhout@student.ru.nl

Abstract. Attribute grammars are a powerful, declarative formalism to implement and reason about programs which, by design, are conveniently modular. Although a full attribute grammar compiler can be tailored to specific needs, its implementation is highly non-trivial, and its long-term maintenance is a major endeavor. In fact, maintaining a traditional attribute grammar system is such a big effort that most systems that were proposed in the past are no longer active. Our approach to implementing attribute grammars is to write them as first-class citizens of a modern functional programming language. We improve a previous zipper-based attribute grammar embedding making it non-intrusive (i.e. no changes to the user-defined data types are required) and type-safe. On top of that, we achieve clearer syntax by using modern Haskell extensions. We believe that our embedding can be employed in practice to implement elegant, efficient, and modular solutions to real-life programming challenges.

Keywords: Embedded Domain Specific Languages · Zipper data structure · Memoization · Attribute Grammars · Higher-Order Attribute Grammars · Functional Programming

1 Introduction

Attribute Grammars (AGs) are a declarative formalism which was proposed by Knuth [8] in the late 60s and allows one to implement and reason about programs in a modular and convenient way. A concrete AG relies on a context-free grammar to define the syntax of a language, and on attributes associated with the productions of the grammar to define the semantics of that language. AGs have been used in practice to specify real programming languages, like for example Haskell [3], as well as powerful pretty printing algorithms [18], deforestation techniques [5], and powerful type systems [12].

When programming with AGs, modularity is achieved due to the possibility of defining and using different aspects of computations as separate attributes.

Attributes are distinct computation units, typically quite simple and modular, that can be combined into elaborated solutions to complex programming problems. They can also be analyzed, debugged, and maintained independently which eases program development and evolution.

AGs have proven to be particularly useful to specify computations over trees: given one tree, several AG systems such as [15, 4, 9, 20] take specifications of which values or attributes need to be computed and perform these computations. The design and coding efforts put into the creation, improvement, and maintenance of these AG systems, however, is tremendous which is often an obstacle to achieving the success they deserve.

An increasingly popular alternative approach to the use of AGs relies on embedding them as first-class citizens of general purpose programming languages [13, 10, 14, 16, 21, 2]. This avoids the burden of implementing a totally new language and associated system by hosting it in state-of-the-art programming languages. Following this approach one then exploits the modern constructions and infrastructure that are already provided by those languages and focuses on the particularities of the domain specific language being developed.

Functional zipper [7] is a powerful abstraction which greatly simplifies the implementation of traversal algorithms performing a lot of local updates. Functional zippers have successfully been applied to construct an AG embedding in Haskell [10, 11]. Despite its elegance, this solution had a major drawback which prevented its use in real-world applications: attributes were not cached, but rather repeatedly recomputed which severely hurt performance. Recently, this flaw has been eliminated [6] and replaced with a different one: the approach became intrusive, i.e. to benefit from the embedding user-defined data structures have to be adjusted.

In this paper we present an alternative mechanism to cache attributes based on a self-organising infinite grid. This graph is laid on top of the user-defined algebraic data type (ADT) and mirrors its structure. The used-defined data type itself remains untouched. The embedding is then based on two (rather than one) coherent zippers traversing the data structures in parallel. On top of being non-intrusive our solution is completely type-safe. Modern Haskell extensions such as `ConstraintKinds` allow us to propagate constraints down in the ADT completely eliminating run-time type casts present in the previous versions. Another side benefit of using modern Haskell features is a cleaner syntax with less code being generated by means of Template Haskell.

2 Running example

In order to illustrate the convenience of our embedding, we will consider the problem of analyzing the semantic correctness of certain functional programming excerpts. In particular, we will focus on **`let`** expressions that are common in functional languages like Haskell [], ML [], or Scala []. Throughout the paper, we shall refer to these expressions as programs in the LET language.

A program in the LET language consists of instruction blocks, where each instruction either: i) declares a variable or ii) defines a nested block. When declaring a variable, the programmer must also assigned a value to it; such value may be a constant, the value of a variable or an expression on these elements. Ultimately, each block defines a value as an expression on variables defined in it.

Examples of programs in LET are given next.

```
program₁ = let x = 4 in x

program₂ = let x = 4 in x + 3

program₃ =
  let x = 4
      y = let w = 2 in x + w
  in  x + y
```

The values associated with `program₁` and `program₂` are, quite straightforwardly, `4` and `7`, respectively. As of `program₃`, its value of `10` is calculated adding the value of `x`, which is `4` and the value of `y`, i.e., `6`. The value of `y` is obtained by adding the value of `w`, which is `2`, and the value of `x`, which, again, is `4`.

Our goal is to implement a semantic analyzer that deals with the scope rules of the LET language. These rules are quite natural and can be described as:

1. if a variable is used in a block, it must be declared in that same block or in an outer one. The declaration of a variable, however, may occur after its use;
2. a variable identifier may be declared at most once within the same block. In an inner block, declaring a variable that has already been declared in an outer one is allowed: the identifier in the local scope hides the definition of the same identifier in the global one.

Let us now consider a more complex LET program:

```
programₑ =
  let x = y
      a = let y = 4 in y + w
      x = 5
      y = 6
  in  x + a
```

According to the scope rules that we have just defined, `programₑ` contains two errors: 1) at the outer block, variable `x` has been declared twice and 2) at the inner block, the use of variable `w` has no binding occurrence at all. Notice that `y` has been declared at both the inner and the outer levels, which in itself is not a problem (the inner declaration hides the outer one).

Programs such as the ones we have presented describe the basic block-structure found in many languages, with the peculiarity that variables can be used before they are defined.

We aim to implement a program that analyses LET programs and computes a list containing the identifiers which do not obey the scope rules of the language. In order to facilitate the debugging phase, we require that the list of invalid identifiers follows the sequential structure of the program. That is to say, e.g., that the semantic meaning of processing $\texttt{program}_\texttt{e}$ is `[w, x]`: the use of the undeclared variable `w` occurs in line 3 whereas the duplicate declaration of `x` occurs in line 4.

Since variables can be used before they are declared, a natural way to implement such an analysis is to traverse a LET program twice: once to accumulate the declarations of identifiers (at each block), and again to check the uses of identifiers against such declarations. The uniqueness of declarations can already be detected in the first traversal: for each newly encountered declaration, it is possible to check whether its identifier has already been declared (in the same level).

In order to implement such program, we first need a representation for programs in the LET language. For this, we may use the following Haskell data-types:

```
type Var    = String
data Let    = Let Decls Expr
data Decls  =  Empty
            |  Cons    Var Expr  Decls
            |  Nested  Var Let   Decls
data Expr   =  Const     Int
            |  Variable  Var
            |  Plus   Expr Expr
            |  Times  Expr Expr
```

In this representation, $\texttt{program}_\texttt{e}$ above is defined as:

```
programe = Let
  (Cons
    "x"
    (Variable "y")
    (Nested
      "a"
      (Let (Cons "y" (Const 4) Empty) (Plus (Variable "y") (Variable "w")))
      (Cons "x" (Const 5) (Cons "y" (Const 6) Empty))
    )
  )
  (Plus (Variable "x") (Variable "a"))
```

Now, we implement name analysis on an abstract LET tree using a set of composable functions.

The function that implements the first traversal described above needs to pass all the information that is required for the second traversal. Namely, in order to compute the final list of errors in the desired order, the second traversal needs to *where* errors have occurred during the first. Also, the value of the nesting

level must also be carried around because it is on the second traversal that the first traversal on nested expressions starts (the initial environment of an inner block is composed by the complete environment of its outer one).

In order to make available all the information that the second traversal needs, the first traversal will build a structure such as:

```
data Let_2   = Let_2 Decls_2 Expr
data Decls_2 = Empty_2
             | Cons_2 Errors Expr Decls_2
             | Nested_2 Errors Lev Let Decls_2
```

We are now ready to implement the two traversal strategy that we have described.

```
type Errors  = [String]
type Lev     = Int

semantics :: Let → Errors
semantics program = errors
 where
  (let_2 , env) = duplicate_Let  program [] 0
  errors        = missing_Let  let_2 env

duplicate_Let  :: Let → [(Var, Lev)] → Lev → (Let_2, [(Var, Lev)])
duplicate_Let  (Let decls expr) dcli lev = (Let_2 decls_2 expr, dclo)
  where (decls_2 , dclo) = duplicate_Decls  decls dcli lev

duplicate_Decls  :: Decls → [(Var, Lev)] → Lev → (Decls_2, [(Var, Lev)])
duplicate_Decls  Empty dcli lev = (Empty_2, dcli)
duplicate_Decls  (Cons var expr decls) dcli lev =
  (Cons_2 error expr decls_2 , dclo)
 where
  error           = if (var, lev) `elem` dcli then [var] else []
  (decls_2 , dclo) = duplicate_Decls  decls ((var, lev) : dcli) lev
duplicate_Decls  (Nested var nested decls) dcli lev =
  (Nested_2 error (lev + 1) nested decls_2 , dclo)
 where
  error           = if (var, lev) `elem` dcli then [var] else []
  (decls_2 , dclo) = duplicate_Decls  decls ((var, lev) : dcli) lev

missing_Let  :: Let_2 → [(Var, Lev)] → Errors
missing_Let  (Let_2 decls expr) env = errors_1 ++ errors_2
 where
  errors_1 = missing_Decls  decls env
  errors_2 = missing_Expr  expr env
```

```
missing_Decls :: Decls_2 → [(Var, Lev)] → Errors
missing_Decls (Cons_2 error expr decls) env = error ++ errors
  where errors = missing_Expr expr env ++ missing_Decls decls env
missing_Decls (Nested_2 error lev nested decls) env = error ++ errors
 where
  (nested_2 , dclo) = duplicate_Let nested env lev
  errors            = missing_Let nested_2 dclo ++ missing_Decls decls env
missing_Decls Empty_2 _ = []

missing_Expr :: Expr → [(Var, b)] → Errors
missing_Expr (Const _) _ = []
missing_Expr (Plus expr_1 expr_2) env =
  missing_Expr expr_1 env ++ missing_Expr expr_2 env
missing_Expr (Times expr_1 expr_2) env =
  missing_Expr expr_1 env ++ missing_Expr expr_2 env
missing_Expr (Variable var) env = if var `elem` map fst env then [] else [var]
```

Notice that $\texttt{duplicate}_{Let}$ not only computes the total environment (using an initially empty accumulating parameter), but it also computes a Let_2 intermediate data structure that stores, e.g., the duplicated variables detected during the first traversal. The second traversal starts with a call to $\texttt{missing}_{Let}$ giving that computed data structure and the accumulated environment as arguments. It produces the list of errors that follows the sequential structure of the program.

In function $\texttt{duplicate}_{Decls}$, for every block we compute: its environment, its 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 (annotated with the level of the block). The level indicates the nesting depth of a block. Observe that we have to distinguish between the same identifier declared at different levels, which is valid.

Finally, please note that in the second traversal of a nested expression, in function $\texttt{missing}_{Decls}$ for the constructor $Nested_2$, the program performs the two traversals to the body of that expression: calls $\texttt{duplicate}_{Let}$ and $\texttt{missing}_{Let}$.

Although the semantic analysis we have implemented for the LET language is relatively simple, still we had to face some challenges.

Indeed, scheduling computations was by no means trivial, with intermingled recursive functions. Also, we had to carefully design and implement intermediate data structures in order to convey data between traversals. These challenges are common to functional programming solutions to realistic programming problems.

In lazy (functional) programming languages, one can avoid both the need for scheduling and for additional data structures by constructing circular programs [**?**]. This strategy, however, compromises the much desired modular nature of the implementations.

In our work, we seek an elegant and efficient alternative to the construction of functional programs.

3 Functional Zippers

Zipper is a data structure commonly used in functional programming for traversal with fast local updates. The zipper data structure was originally conceived by Huet[7] in the context of trees. We will, however, first consider a simpler problem: a bidirectional list traversal.

Lists Suppose that we would like to update a list at a specific position:

```
modify :: (a → [a]) → Int → [a] → [a]
modify f i xs = helper [] xs 0
 where helper before (x : after) !j
         | j ≡ i    = before ++ f x ++ after
         | otherwise = helper (before ++ [x]) after (j + 1)
       helper _ [] _ = error "Index out of bounds."
```

Here `modify` takes an update action `f`[6], an index `i`, and a list `xs` and returns a new list with the `i`'th element replaced with the result of `f`.First, we "unpack" the list into `before` = $[xs_0, \ldots xs_{i-1}]$, `x` = xs_i, and `after` = $[xs_{i+1}, \ldots]$. Then replace `x` by `f(x)` and finally "pack" the result back into a single list. If we do a lot of updates, we end up unpacking and packing the list over and over again – very time-consuming for long lists. We would thus like to benefit from fusion without explicitly working with the unpacked representation as it is bug-prone. A list zipper provides a way to achieve this.

A zipper consists of a focus (alternatively called a hole) and surrounding context[7]:

```
data Zipper a  = Zipper { _hole :: a, _cxt :: !(Context a) }
data Context a = Context [a] [a]
```

where the *Context* keeps track of elements to the left (`before` in `modify`) and to the right (`after` in `modify`) of the focus. We can now define movements:

```
left :: Zipper a → Maybe (Zipper a)
left (Zipper _    (Context [] _))        = Nothing
left (Zipper hole (Context (l : ls) rs)) = Just $
  Zipper l $ Context ls (hole : rs)

right :: Zipper a → Maybe (Zipper a)
right (Zipper _    (Context _ []))        = Nothing
right (Zipper hole (Context ls (r : rs))) = Just $
  Zipper r $ Context (hole : ls) rs
```

[6] `f` returns a list rather than a single element to prevent curious readers from suggesting to use a boxed array instead of a list.

[7] '`!`' in front of the type of `_cxt` is a strictness annotation enabled by the `BangPatterns` extension. Context can be seen as a path from the to the current focus and is always finite, i.e. there is no reason for it to be lazy.

and functions for entering and leaving the zipper:

```
enter :: [a] → Maybe (Zipper a)
enter []       = Nothing
enter (x : xs) = Just $ Zipper x (Context [] xs)

leave :: Zipper a → [a]
leave (Zipper hole (Context ls rs)) = reverse ls ++ hole : rs
```

Finally, we define a local version of our `modify` function (**TODO:***Boy, is this function ugly...*)

```
modify :: (a → [a]) → Zipper a → Maybe (Zipper a)
modify f (Zipper hole (Context ls rs)) = case f hole of
  (x : xs) → Just $ Zipper x (Context ls (xs ++ rs))
  []       → case rs of
    (r : rs') → Just $ Zipper r (Context ls rs')
    []        → case ls of
      (l : ls') → Just $ Zipper l (Context ls' rs)
      []        → Nothing
```

using which we can perform multiple updates efficiently and with minimal code bloat[8]:

```
modifyExample :: IO ()
modifyExample = print $
  enter @Int >=> right
             >=> right
             >=> modify (const [])
             >=> modify (return ∘ (+1))
             >=> left
             >=> modify (return ∘ negate)
             >=> return ∘ leave $ [1, 2, 3, 4, 5]
```

Trees Consider now a binary tree data structure:

```
data Tree a = Leaf !a | Fork (Tree a) (Tree a)
```

A binary tree zipper is slightly more insteresting than the list zipper, because we can move up and down the tree as well as left and right. The zipper again consists of a hole (a subtree we are focused on) and its surrounding context (a path from the hole to the root of the tree):

[8] Operator `>=>` comes from *Control.Monad* module in `base` and has the following signature:

```
(>=>) :: Monad m => (a → m b) → (b → m c) → a → m c
```

```
data Zipper a  = Zipper { _hole :: Tree a, _cxt :: !(Context a) }
data Context a = Top
               | Left !(Context a) (Tree a)
               | Right (Tree a) !(Context a)
```

To move the zipper down, we "unpack" the current hole:

```
down :: Zipper a → Maybe (Zipper a)
down (Zipper (Leaf _)   _)   = Nothing
down (Zipper (Fork l r) cxt) = Just $ Zipper r (Right l cxt)
```

Context stores everything we need to reconstruct the hole, and `up` does exactly that:

```
up :: Zipper a → Maybe (Zipper a)
up (Zipper _ Top) = Nothing
up (Zipper l (Left cxt r)) = Just $ Zipper (Fork l r) cxt
up (Zipper r (Right l cxt)) = Just $ Zipper (Fork l r) cxt
```

Implementations of `left`, `right`, and `enter` are very similar to the list zipper case and are left as an exercise for the reader. `leave` differs slightly in that we now move all the way up rather than left:

```
leave :: Zipper a → Tree a
leave z = case up z of
  Just z' → leave z'
  Nothing → _hole z
```

Generic Zipper The list and binary tree zippers we have considered so far are homogeneous zippers: the type of the focus does not change upon zipper movement. Such a zipper can be a very useful abstraction. For example, a well-known window manager XMonad[17] uses a rose tree zipper to track the window under focus. For other tasks, however, one might need to traverse heterogeneous structures. A zipper that can accomodate such needs is usually called a *generic zipper* as it relies only on the generic structure of Algebraic Data Types (ADTs). One can view an ADT as an Abstract Syntax Tree (AST) where each node is a Haskell constructor rather than a syntax construct.

The generic zipper we will use is very similar to the one presented in[1]. The most common technique in Haskell for supporting heterogeneous types is Existential Quantification. However, not every type can act as a hole. To support moving down the tree, we need the hole to be *dissectible*, i.e. we would like to be able to dissect the value into the constructor and its arguments. Even though *Data.Data*.`gfoldl` allows us to achieve this, we define out own typeclass which additionally allows us to propagate down arbitrary constraints:

```
class Dissectible (c :: Type → Constraint) (a :: Type) where
  dissect :: a → Left c a
```

```
data Left c expects where
  LOne  :: b → Left c b
  LCons :: (c b, Dissectible c b)
        => Left c (b → expects) → b → Left c expects
```

For example, here is how we can make *Tree* an instance of *Disssectible*

```
instance c (Tree a) => Dissectible c (Tree a) where
  dissect (Fork l r) = LOne Fork `LCons` l `LCons` r
  dissect x = LOne x
```

We can unpack a *Fork* and the zipper will thus be able to go down. *Leaf*s, however, are left untouched and trying to go down from a *Leaf* will return *Nothing*.

To allow the zipper to move left and right, we need a means to encode arguments to the right of the hole. Following Adams et al, we define a GADT representing constructor arguments to the right of the hole:

```
data Right c provides r where
  RNil  :: Right c r r
  RCons :: (c b, Dissectible c b)
        => b → Right c provides r → Right c (b → provides) r
```

For example, for a tuple *(Int, Int, Int, Int, Int, Int)*, we can have

```
lefts = LOne (,,,,,) `LCons` 1 `LCons` 2 `LCons` 3
hole = 4
rights = 5 `RCons` 6 `RCons` RNil
```

generalising this a little, we arrive at:

```
data LocalContext c hole rights parent =
  LocalContext !(Left c (hole → rights)) !(Right c rights parent)

data Context :: (Type → Constraint) → Type → Type → Type where
  RootContext :: ∀ c root. Context c root root
  (:>) :: ∀ c parent root hole rights. (c parent, Dissectible c parent)
       => !(Context c parent root)
       → !(LocalContext c hole rights parent)
       → Context c hole root
```

And just like before the *Zipper* is a product of the hole and context:

```
data Zipper (c :: Type → Constraint) (root :: Type) =
  ∀ hole. (c hole, Dissectible c hole) =>
    Zipper { _zHole :: !hole
           , _zCxt  :: !(Context c hole root)
           }
```

Implementation of movements is quite straightforward and is left out. Please, refer to (**TODO:***github repo*) for complete code. We now consider a rather interesting application of generic zipper: embedding of attribute grammars.

4 Attribute Grammars

Attribute grammars (AGs) are an extension of context-free grammars that allow to specify context-sensitive syntax as well as the semantics. AGs achieve it by associating a set of attributes with each grammar symbol. These attributes are defined using evaluation rules assiciated with production rules of the context-free grammar.

Attributes are then usually divided into two disjoint sets: synthesized attributes and the inherited attributes. Such distinction is required for the construction of a dependency graph. It is then used for specification of the evaluation order and detection of circularity. In the zipper-based embedding of attribute grammars we make no use of a dependency graph and thus do not divide attributes into classes.

Let us consider the repmin problem as an example of a problem that requires multiple traversals:

> Given a tree of integers, replace every integer with the minimum integer in the tree, in one pass.

The classical solution is the following circular program:

```
repmin :: Tree Int → Tree Int
repmin t = t'
  where (t', m') = go t m'
        go (Leaf x    ) m = (Leaf m, x)
        go (Fork xs ys) m = (Fork xs' ys', min m_x m_y)
          where (xs', m_x) = go xs m
                (ys', m_y) = go ys m
```

Although quite elegant, the code lacks modularity and is very difficult to reason about. Attribute Grammars provide a more modular approach. Viera et al[22] identified three steps for solving repmin: computing the minimal value, passing it down from the root to the leaves, and constructing the resulting tree. We can associate each step with an attribute[19]:

- A synthesized attribute `localMin :: Int` represents the minimum value of a subtree. Computing the minimal value thus corresponds to evaluation of the `localMin` attribute for the root tree.
- An inherited attribute `globalMin :: Int` is used to pass down the minimal value.
- Finally, a synthesized attribute `updated :: Tree Int` is the subtree with leaf values replaced by values of their `globalMin` attributes. The solution is thus the value of `updated` attribute for the root tree.

The obtained AG is presented in figure 1. **TODO:***Do we actually need to explain the algorithms here? It seems a little bit childish to explain how to compute the minimum of a binary tree... If we absolutely have to explain stuff, maybe just put it in the caption.*

```
SYN Tree Int [localMin : Int]
SEM Tree Int | Leaf lhs.localMin = @value
             | Fork lhs.localMin = min @left.localMin
                                       @right.localMin

SYN Tree Int [updated : Tree Int]
SEM Tree Int | Leaf lhs.updated = Leaf @lhs.globalMin
             | Fork lhs.updated = Fork @left.updated
                                       @right.updated


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

DATA Root | Root tree : Tree Int
SEM  Root | Root tree.globalMin = @tree.localMin
```

Fig. 1: Attribute grammar for repmin. The syntax is closely mirrors the one used in[19]. `SYN` and `INH` introduce synthesized and inherited attributes respectively. `SEM` is used for defining semantic rules. A new data type *`Root`* is introduced as it is common in the AG setting to "connect" `localMin` with `globalMin`.

We now move on to embed this attribute grammar into Haskell. Semantic rules simply become functions, which, given a zipper, return values of the attributes. For example,

```
localMin :: Zipper (WhereAmI Position) (Tree Int) → Int
localMin z@(Zipper hole _) = case whereami hole of
  C_Leaf  → let Leaf x = hole in x
  C_Fork  → let Just l = child 0 z; Just r = child 1 z
            in min (localMin l) (localMin r)
```

Apart from the type signature, the code is pretty straightforward and closely mirrors the AG we defined earlier. `whereami` function allows us to "look around" and returns the position of the zipper. Thus the **`case`** corresponds to the pattern matches on the left of the vertical bars on figure 1.

Position of the zipper is encoded using the following GADT which can be generated automatically using Template Haskell:

```
data Position :: Type → Type where
  C_Leaf  :: Position (Tree Int)
  C_Fork  :: Position (Tree Int)
```

Parametrization on the type of the hole allows the code like **`let`** *`Leaf`* `x = hole` **`in`** `x` to typecheck, even though the generic zipper itself knows close to nothing about

the type of the hole. It might seem trivial at first, because the binary tree zipper is in fact homogeneous. The "position trick" however extends also to heterogeneous zippers which we will encounter in more advanced examples.

```
child :: Int → Zipper cxt root → Maybe (Zipper cxt root)
```

`child n` moves the zipper to the `n`'th child, if there is one.

5 Related Work

6 Conclusion

Acknowledgements

References

1. Adams, M.D.: Scrap your zippers: A generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming. pp. 13–24. WGP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863495.1863499, http://doi.acm.org/10.1145/1863495.1863499
2. Balestrieri, F.: The Productivity of Polymorphic Stream Equations and The Composition of Circular Traversals. Ph.D. thesis, University of Nottingham (2015)
3. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Haskell Symposium. pp. 93–104 (2009)
4. Dijkstra, A., Swierstra, D.: Typing Haskell with an Attribute Grammar (Part I). Tech. Rep. UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University (2004)
5. Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: Symposium on Partial Evaluation and Program Manipulation. pp. 102–111. ACM (2007)
6. Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M.: Memoized zipper-based attribute grammars and their higher order extension. Science of Computer Programming (2018)
7. Huet, G.: The zipper. Journal of functional programming **7**(5), 549–554 (1997)
8. Knuth, D.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (June 1968), *Correction: Mathematical Systems Theory* 5 (1), March 1971.
9. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In: International Conference on Compiler Construction. pp. 298–301. Springer-Verlag (1998)
10. Martins, P., Fernandes, J.P., Saraiva, J.: Zipper-based attribute grammars and their extensions. In: Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings. pp. 135–149 (2013)
11. Martins, P., Fernandes, J.P., Saraiva, J., Wyk, E.V., Sloane, A.: Embedding attribute grammars and their extensions using functional zippers. Science of Computer Programming **132**, 2 – 28 (2016), selected and extended papers from SBLP 2013

12. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In: International Conference on Generative Programming. pp. 43–52. ACM (2010)
13. de Moor, O., Backhouse, K., Swierstra, D.: First-Class Attribute Grammars. In: 3rd. Workshop on Attribute Grammars and their Applications. pp. 1–20. Ponte de Lima, Portugal (2000)
14. Norell, U., Gerdes, A.: Attribute Grammars in Erlang. In: Workshop on Erlang. pp. 1–12. 2015, ACM (2015)
15. Reps, T., Teitelbaum, T.: The synthesizer generator. SIGPLAN Not. **19**(5), 42–48 (Apr 1984)
16. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electronic Notes in Theoretical Computer Science **253**(7), 205–219 (2010)
17. Stewart, D., Sjanssen, S.: Xmonad. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 119–119. Haskell '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1291201.1291218, `http://doi.acm.org/10.1145/1291201.1291218`
18. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In: Third Summer School on Advanced Functional Programming. LNCS Tutorial, vol. 1608, pp. 150–206. Springer Verlag (1999)
19. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. In: International School on Advanced Functional Programming. pp. 150–206. Springer (1998)
20. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electronic Notes in Theoretical Computer Science **203**(2), 103–116 (2008)
21. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: International Conference on Functional Programming. pp. 245–256. ACM (2009)
22. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 245–256. ICFP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1596550.1596586, `http://doi.acm.org/10.1145/1596550.1596586`