Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Emanuel Silva Rodrigues

On the performance of Strategic Attribute Grammars

August 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Emanuel Silva Rodrigues

On the performance of Strategic Attribute Grammars

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
João Saraiva

August 2022

STATEMENT OF INTEGRITY

ABSTRACT

Strategic programming is a powerful technique used in language processing to define functions that traverse abstract syntax trees. With strategies, the programmer only indicates the nodes of the tree where the work has to be done, and the strategy used to traverse the whole tree and apply the function that works only on the defined nodes.

In Haskell, there are two libraries that implement strategies: Strafunski and an equivalent library developed by DI: Ztrategic. Beyond that, we also have the Kiama library which is implemented in the Scala programming language. The Ztrategic library uses memorization in order to save work. Using memorization, the elimination of all occurrences of "bad smells" in an abstract tree of a program is done only once!

In this thesis, we present a detailed study of the performance of the Kiama, Ztrategic, and memoized Ztrategic libraries, using different strategic problems and input languages.

KEYWORDS Strategic programming, Attribute grammars, Zippers, Ztrategic.

RESUMO

Programação estratégica é uma técnica poderosa usada em processamento de linguagens para definir funções que atravessam árvores de sintaxe abstracta. Com estratégias o programador apenas indica os nodos da árvore onde o trabalho tem de ser feito, e depois que estratégia é utilizada para atravessar toda a árvore e aplicar a função que faz trabalho apenas nos nodos definidos.

Em Haskell existem duas bibliotecas de combinadores que implementam estratégias: Strafunski e uma biblioteca equivalente desenvolvida no DI: Ztrategic. Existe também outra biblioteca desenvolvida em Scala, Kiama. A biblioteca Ztrategic usa memorização de modo a poupar trabalho. Usando memorização, a eliminação de todas a ocorrências do "mau cheiro" numa árvore abstracta de um programa é feita apenas uma vez!

Nesta tese faz-se um estudo detalhado da performance das bibliotecas Kiama, Ztrategic, e memoized Ztrategic, utilizando diferentes problemas de programação estratégica e diferentes linguagens de input.

PALAVRAS-CHAVE Programação estratégica, Gramáticas de atributos, Zippers, Ztrategic.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1 INTRODUCTION

Since Algol was designed in the '60s, as the first high-level programming language (van Wijngaarcien et al., 1977), languages have evolved dramatically. In fact, modern languages offer powerful syntactic and semantic mechanisms that improve programmers productivity. In response to such developments, the software language engineering community also developed advanced techniques to specify such new mechanisms.

Strategic term rewriting (Luttik and Visser, 1997) and Attribute Grammars (AG) (Knuth, 1968) have a long history in supporting the development of modern software language analysis, transformations, and optimizations. The former relies on *strategies* (recursion schemes) to traverse a tree while applying a set of rewrite rules, while the latter is suitable to express context-dependent language processing algorithms. Many language engineering systems have been developed supporting both AGs (Gray et al., 1992; Reps and Teitelbaum, 1984; Kuiper and Saraiva, 1998; Mernik et al., 1995; Ekman and Hedin, 2007; Dijkstra and Swierstra, 2005; Van Wyk et al., 2008) and rewriting strategies (van den Brand et al., 2001; Balland et al., 2007; Lämmel and Visser, 2002; Cordy, 2004; Sloane et al., 2010; Visser, 2001). These powerful systems, however, are large systems supporting their own AG or strategic specification language, thus requiring a considerable development effort to extend and combine.

A more flexible approach is obtained when we consider the embedding of such techniques in a general purpose language. Language embeddings, however, usually rely on advanced mechanisms of the host language, which makes them difficult to combine. For example, Strafunski (Lämmel and Visser, 2002) offers a powerful embedding of strategic term rewriting in Haskell, but it can not be easily combined with the Haskell embedding of AGs as provided in (de Moor et al., 2000; Martins et al., 2013). The former works directly on the underlying tree, while the latter on a *zipper* representation of the tree.

1.1 ZIPPING STRATEGIES AND ATTRIBUTE GRAMMARS

More recently, an embedding of both strategic tree rewriting and attribute grammars in a zipper-based, purely functional setting as been defined (Macedo et al., 2022b). This embedding relies on generic zippers (Huet, 1997), which is a simple generic tree-walk mechanism to navigate on both homogeneous and heterogeneous data structures. Traversals on heterogeneous data structures (i.e., data structures composed of different data structures) is the main ingredient of both strategies and AGs. Thus, zippers provide the building block mechanism we will reuse for expressing the purely-functional embedding of both techniques. The embedding of the two techniques in the same setting has several advantages: First, we easily combine/zip attribute grammars and strategies, thus providing language engineers the best of the two worlds. Second, the combined embedding is easier to maintain and extend since it is written in a concise and uniform setting. This results in a very small library (200 lines of Haskell code) which is able to express advanced (static) analyses and transformation tasks.

This concise and multiple paradigm embedding, however, has a key drawback: has shown in (Fernandes et al., 2019) the resulting AG evaluators re-computes the same attribute instances. As a consequence, those evaluators do not provide a proper embedding of the AG formalism, which, as expected, result in very inefficient evaluators. In (Fernandes et al., 2019) memoization of attribute instances has been incorporated to the zipper-based embedding of AGs, which does improve the evaluators runtime.

In order to combine strategic term re-writing, as presented in (Macedo et al., 2022b), with the runtime efficient memoized AGs, proposed in (Fernandes et al., 2019), we recently re-wrote the zipper-based strategic library so that it works on memoized zipper-based trees (Macedo et al., 2022a) To validate our new embedding of strategic AGs we need both to access its expressiveness and its performance. Thus, we would like to provide answers to the following research questions.

1.2 RESEARCH QUESTIONS

- *Research Question 1:* Is the combined memoized embedding of Strategic AGs expressive enough to specify complex language engineering algorithms?
- *Research Question 2:* How does the performance of memoized strategic AGs compare to the state-of-the-art defined by Kiama (Sloane et al., 2010)?

In order to answer the first question, we will consider two different language engineering problems: First, we will consider the let (sub)language that is part of most functional programming languages. We will consider two usual language processor tasks: name analysis and simple (arithmetic) optimizations. Moreover, we will express in the Haskell embedding a large and complex optimal pretty printing algorithm, which allows multiple layouts, and it does perform four traversal over its own data structure to generate the pretties print (Swierstra et al., 1999).

Finally, to answer the third research question, we will consider both the let processor and the optimal pretty printing algorithm. Moreover, to compare the Haskell embedding to the Kiama we will implement the complex pretty printing algorithm in Kiama, as well.

1.3 STRUCTURE OF THE THESIS

This work describes some libraries for strategic programming and compares them with various examples. More details of what are attribute grammars and zippers are found in chapter 2, as well as some libraries used for strategic programming. Chapter 3 presents some examples used to benchmark the performance of the different libraries. Conclusions can be found on chapter 4.

2

STATE OF THE ART

2.1 STRAFUNSKI

Strafunski (Lämmel and Visser, 2003) is a Haskell strategic programming library that makes use of generics to traverse a data structure. Strategies are made using function combinators. The combinators are qualified as TP (type preserving) or TU (type unifying) which represent transformations or reductions respectively.

2.2 ATTRIBUTE GRAMMARS

An attribute grammar is a way to denote syntax trees with semantic attributes. These attributes are associated with terminal and non-terminal symbols.(Kramer and Van Wyk, 2020)

If we consider the repmin problem, to transform a binary leaf tree of integers into a new one of the same shape but with all the leaf values replaced by the minimum leaf value of the original tree, the grammar for this problem can be seen as the data type of the tree.

In Haskell, we consider the following data type:

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

Listing 2.1: Repmin data type

With attribute grammars, there are two types of attributes, synthesized and inherited. A synthesized attribute is calculated from the values of the attributes of the children, and an inherited attribute is obtained by using the attribute values of the parent or siblings. Synthesized attributes serve to pass information up the tree, while inherited ones allow values to be passed down.(Knuth, 1968)

Therefore, to compute the minimum value for the repmin problem, we define the synthesized attribute *lm* (local minimum) associated with the non-terminal Tree.

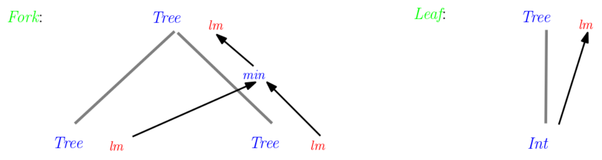

Figure 1: *lm* attribute

As we can see on figure 1, in the Fork production, the *min* function gets as arguments the synthesized minimum of the two subtrees, in order to synthesize the minimum of the tree. For the Leaf production, the minimum is as defined as the Int value stored in there,

With the minimum value synthesized, to compute the new tree, we will need to pass it downwards. For that, we create an inherited attribute, *gm* (global minimum) which is inherited by Tree nodes.

In figure 2, we see that the attributes are just copied downwards in the tree.

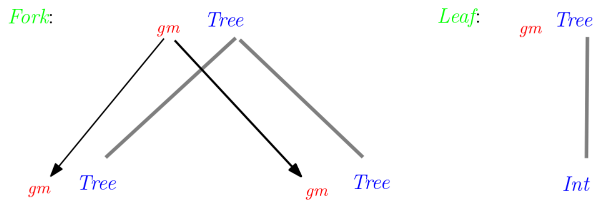

Figure 2: *gm* attribute

Kiama and Ztrategic both make use of attribute grammars along with strategic term rewriting. In particular, Ztrategic uses a library of attribute grammars in Haskell developed at DI(Fernandes et al., 2019), which was extended with memoization.

2.3 ZIPPERS

Zippers were conceived to represent a tree, along with a subtree which is the focus of our attention. During computation, the focus may move within the tree. Manipulation of a zipper is achieved through a set of functions that allow access to all nodes of the tree.(Martins et al., 2013)

Generic zippers work for both homogeneous and heterogeneous data types and are available as a Haskell library. They provide a uniform way to navigate in heterogeneous data structures without needing to distinguish which nodes are being traversed. Any data type with instances of Data and Typeable type classes can be navigated using generic zippers.

In Haskell, to navigate using zippers, first we need to use the function *toZipper*, which given a data type returns a zipper of that. To traverse the zipper we can functions like `down' :: Zipper a -> Maybe (Zipper a)`, that goes down to leftmost child and returns Nothing if no child exists. In order to get the node that we are focusing on with the zipper, we use the function `getHole :: Typeable b =>Zipper a -> Maybe b`.

In addition to the generic zippers library, we implemented some simple combinators, that make it easier to write zipper-based functions. More exactly, we have defined the following:

- (.$) - access the child of a node given its index (.$1 represents the first child)
- parent - moves the focus to the parent of a node
- (.$<), (.$>) - moves to the left or right n th sibling, respectively (.$<1 moves to the left sibling)
- (.|) - checks whether the current location is a sibling of a tree, or not

2.4 ZTRATEGIC

Ztrategic is a library which embeds strategic term rewriting and attribute grammars together with generic zippers. Its API is based on Strafunski's API but with the added possibility of manipulating the zippers that traverse the tree. Likewise, the strategies in Ztrategic are classified as TP (type preserving) or TU (type unifying).(Macedo et al., 2021) The usage of attribute grammars lets us make use of the grammar context, unlike Strafunski.

Some functions provided by Ztrategic are:

- applyTP/applyTU - Apply a given strategy to the problem tree
- full_tdTP/full_tdTU - top-down traversal of the whole tree
- full_buTP/full_buTU - bottom-up traversal of the whole tree
- adhocTP/adhocTU - Joins two functions into a single one
- adhocTPZ/adhocTPZ - Joins two functions into a single one with access to the zipper
- failTP/failTU - Failing function
- idTP/idTU - Identity function

2.4.1 *Repmin in Ztrategic*

In Ztrategic we will define the previous stated attributes in the following way:

```haskell
type AGTree a = Zipper Tree -> a

globmin :: AGTree Int
globmin t = case constructor t of
            CRoot   -> locmin (t.$1)
            CLeaf   -> globmin (parent t)
            CFork   -> globmin (parent t)

locmin :: AGTree Int
locmin t = case constructor t of
            CLeaf   -> lexeme t
            CFork   -> min (locmin (t.$1))
```

```
                            (locmin (t.$2))
```

Listing 2.2: Repmin attributes

In listing 2.2, we make use of the *constructor* function to look at the zipper and return the node its currently focused on, that attribute is used to define the behaviour for each node in our attributes.

As previously stated, the *locmin* attribute will locate the minimum, using the *lexeme* function when on a Leaf node to extract its value, and when on a Fork the minimum will be the result of *min* function applied to the minimum of its children.

The *lexeme* function returns the value present in the Leaf.

Then, we define the strategy that will be used to solve this problem:

```
repmin :: Tree -> Maybe (Zipper Tree)
repmin t = applyTP (full_tdTP step) $ toZipper t
  where step = failTP `adhocTPZ` aux

aux :: Tree -> Zipper Tree -> Maybe Tree
aux (Leaf _) z = Just $ Leaf (globmin z)
aux _ _ = Nothing
```

Listing 2.3: Repmin Strategy

For the strategy present in listing 2.3, first we convert our tree into a zipper through the function *toZipper* and then using *applyTP* we apply the type preserving strategy to it. With *full_tdTP* we indicate that we want to traverse the zipper in a top-down order and apply at every node the *step* function. This function will make use of *aux* to replace the leaf value with its global minimum and its combined with *failTP* through the use of *adhocTPZ* so that in other cases that aux doesn't cover the strategy fails.

We can also solve this problem without the use of strategies thorugh an attribute that will replace the Leaf value with the globmal minimum.

```
replace :: AGTree Tree
replace t = case constructor t of
            CRoot   -> replace (t.$1)
            CLeaf   -> Leaf (globmin t)
            CFork   -> Fork (replace (t.$1))
                            (replace (t.$2))
```

Listing 2.4: Replace attribute

The *replace* attribute found in listing 2.4 will make use of the *globmin* attribute when in a Leaf node to replace it with its minimum and for the other cases it will call *replace* to the children nodes of where we are, thus navigating the whole tree and applying *globmin* to all the nodes.

Beyond that, we can also solve this problem without the use of attribute grammars, using only strategies. For that, we will define a type unifying strategy that computes the minimum and then build the tree with a type preserving strategy.

```
findMin :: Tree -> Int
findMin t = minimum $ applyTU (full_tdTU step) $ toZipper t
       where step = failTU `adhocTU` nodeVal
             nodeVal :: Tree -> [Int]
             nodeVal (Leaf x) = return x
             nodeVal _        = []


repMin :: Tree -> Tree
repMin t = fromZipper $ fromJust $ applyTP (full_tdTP step) $ toZipper t
 where step = failTP `adhocTP` aux
       v = findMin t
       aux :: Tree -> Maybe Tree
       aux (Leaf r) = Just $ Leaf v
       aux _        = Nothing
```

Listing 2.5: Repmin using only strategies

2.5 KIAMA

Kiama is a library developed in the Scala programming language which embeds strategic term rewriting and attribute grammars. In Kiama attributes are defined as Scala functions, and strategic term rewriting is expressed on Scala data structures via a set of strategic combinators.

Kiama caches attribute values in a global cache, in order to reuse attribute values computed in the original tree that are not affected by the rewriting. However, that induces an overhead to maintain it updated because attribute values discarded by the rewriting process need to be purged from the cache.(Macedo et al., 2021)

2.5.1 *Repmin in Kiama*

In Kiama, the Repmin problem was already defined as an example, in its solution it uses the following data type:

```
sealed abstract class RepminTree extends Product
case class Fork(left : RepminTree, right : RepminTree) extends RepminTree
case class Leaf(value : Int) extends RepminTree
```

Listing 2.6: Repmin data type in Kiama

As with Ztrategic we have the attributes *locmin* and *globmin*, these are implemented in a way similar to the previous implementation in Haskell. To define the attributes, we will use an *attr* block.

```
    val locmin : RepminTree => Int =
        attr {
            case Fork(l, r) => locmin(l).min(locmin(r))
            case Leaf(v)    => v
        }

    val globmin : RepminTree => Int =
        attr {
            case tree.parent(p) =>
                globmin(p)
            case t =>
                locmin(t)
        }
```

Listing 2.7: Repmin attributes in Kiama

With the main attributes implemented, we can solve the problem using strategies or through the use of another attribute. With an attribute solution, we use the repmin attribute, which applies globmin to the Leafs.

```
    val repmin : RepminTree => RepminTree =
        attr {
            case Fork(l, r) => Fork(repmin(l), repmin(r))
            case t : Leaf   => Leaf(globmin(t))
        }
```

Listing 2.8: Repmin attribute

Using a strategic solution, we define a function to apply the attribute *globmin* to a Leaf Node using a rule combinator that will pattern match on the term and then return a new one. The everywhere function is the traversal that we will use on the tree with our *aux* function, this function will apply *aux* in top-down fashion to all nodes of the tree. Other ways to traverse the tree that Kiama lets us use are *everywherebu*, which behaves the same as everywhere but in a bottom-up fashion and innermost, which will try to apply a transformation to all its nodes until it fails.

```
    val aux = {
        rule[RepminTree] {
            case t@Leaf(_) => Leaf(globmin(t))
        }
    }

    val normal = everywhere(aux)

    def rep (r : RepminTree) : RepminTree = {
        rewrite(normal)(r)
    }
```

Listing 2.9: Repmin strategy in Kiama

3

PERFORMANCE OF STRATEGIC ATTRIBUTE GRAMMARS

To compare Ztrategic with Kiama we developed various examples, and then compared and analysed their performance.

3.1 REPMIN

As seen in the previous chapters, we developed the Repmin problem in both Ztrategic and Kiama. With it and its implementations already explained, we benchmarked its performance against our various solutions. To measure it, we used a function to create a tree, ran repmin for that tree and finally, printed the sum of all the nodes of the final tree.

For our benchmarks the repmin version used were the original version without memoization using only attributes (Original) using the attribute in listing 2.4, the memoized version of Original (MemoAG), the version using a type preserving strategy (Ztrategic - TP), as seen on 2.3, and its memoized version (MemoZtrategic - TP), the one using a type unifying strategy to compute the minimum and then a type preserving one to construct the tree (Ztrategic - TU TP), described in listing 2.5 the Kiama version using only attributes (Kiama) and finally, the Kiama version using a strategy (Kiama Strategy), both in listings 2.8 and 2.9 respectively.

In figures 3 and 4 we show the performance of our repmin solutions, using tree sizes of 5000, 10000, 15000, 20000 and 40000, with its runtime in seconds.

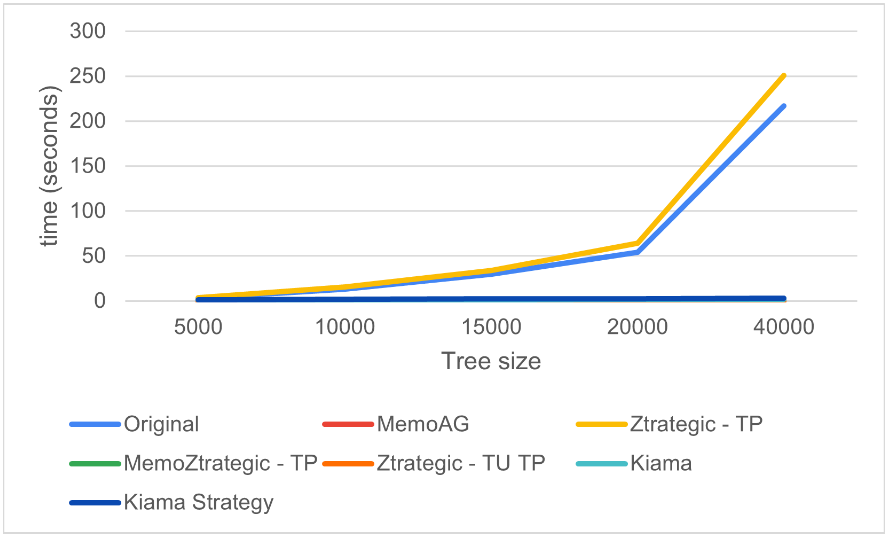

Figure 3: Non-memoized versus Memoized versus Kiama repmin programms

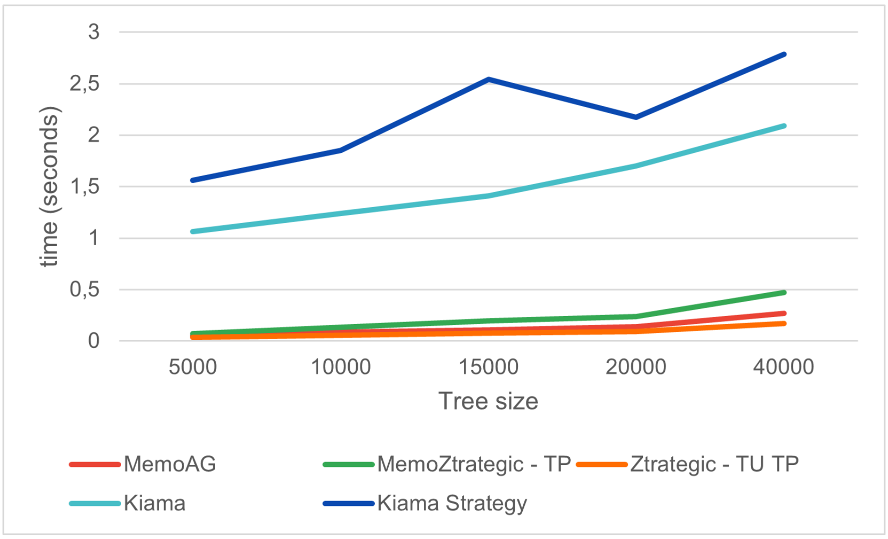

Figure 4: Memoized versus Kiama repmin programms

As seen in the previous figures, the versions of repmin present in figure 4 are all faster than the Original and Ztrategic - TP implementations, this is due to the fact that all of those versions, except Ztrategic - TU TP, cache its attributes. The Ztrategic - TU TP version, the one that doesn't use attributes and only strategies, is the faster as it only traverses the tree 2 times, one to calculate the minimum and another to replace the tree.

3.2 BLOCK

The list based Block language consists of a list of either a declaration of an identifier, the use of an identifier, or a nested block. This language allows abstract modelling of how most programming languages declare and use variables. The declarations of identifiers are not required to occur before their first use. An identifier can only be declared once per nesting level, and only declared names can be used.

3.2.1 *Block in Haskell*

First, define the syntax of the language using the following data types:

```haskell
data P  = Root Its
          deriving (Typeable, Data,Eq)

data Its = ConsIts It Its
         | NilIts
       deriving (Typeable, Data,Eq)

data It = Decl Name
        | Use Name
        | Block Its
        deriving (Typeable, Data,Eq)

type Name = String
```

Listing 3.1: Haskell data types for Block

We use the *It* data type to model an instruction, which can be a declaration, usage, or nested block of instructions. We define a sequence of instructions with the *Its* constructor, and we use the P constructor to represent the start of the abstract syntax tree.

With this example, we aim to implement a function that, for a given Block, produces a list of identifiers containing errors, i.e., using an identifier which was not declared, or double declaration of an identifier.

With this, we start by creating an attribute, *dclo*, that will synthesize the list of declared identifiers of a block. For that function we will also create an inherited attribute, *dcli*, that contains the accumulated declarations thus far.

```haskell
type AGTree a  = Zipper P -> a

dclo :: AGTree Env
dclo t =  case constructor t of
                    CNilIts   -> dcli t
                    CConsIts  -> dclo (t.$2)
                    CDecl     -> (lexeme t,lev t) : (dcli t)
                    CUse      -> dcli t
                    CBlock    -> dcli t

dcli :: AGTree Env
dcli t =  case constructor t of
                    CRoot     -> []
                    CNilIts   -> case  (constructor $ parent t) of
                                                 CConsIts  -> dclo (t.$<1)
```

```
                                            CBlock    -> env (parent t)
                                            CRoot     -> []
                    CConsIts  -> case  (constructor $ parent t) of
                                            CConsIts  -> dclo (t.$<1)
                                            CBlock    -> env (parent t)
                                            CRoot     -> []
                    CBlock    -> dcli (parent t)
                    CUse      -> dcli (parent t)
                    CDecl     -> dcli (parent t)
```

Listing 3.2: Definition of *dclo* and *dcli*

The constructor function peeks into the zipper and returns the node the zipper is focused on - we use it to define different behaviour for different nodes of the tree. The *lexeme* function returns the name of the identifier and *lev* is an inherited attribute that computes the nesting level in which that identifier is.

```
lev :: AGTree Int
lev t =  case constructor t of
                     CRoot     ->  0
                     CBlock    ->  lev (parent t)
                     CUse      ->  lev (parent t)
                     CDecl     ->  lev (parent t)
                     _         ->  case  (constructor $ parent t) of
                                        CBlock    -> (lev (parent t)) + 1
                                        CConsIts  -> lev (parent t)
                                        CRoot     -> 0
```

Listing 3.3: Definition of *lev*

Now we define an inherited attribute, *env*, to obtain the environment (available identifiers) of a block.

```
env :: AGTree Env
env t =  case constructor t of
                    CRoot      ->  dclo t
                    CBlock     ->  env (parent t)
                    CUse       ->  env (parent t)
                    CDecl      ->  env (parent t)
                    _          ->  case (constructor $ parent t) of
                                                CBlock    -> dclo t
                                                CConsIts  -> env (parent t)
                                                CRoot     -> dclo t
```

Listing 3.4: Definition of *env*

With these attributes, we can now synthesise the list of errors. Only two constructors contribute to that list, *Decl* and *Use*. In declarations, the identifier **must not be in** the accumulated list of declarations on the same level. In uses of identifiers, it **must be in** the environment of its block. Thus, to create the *errors* attribute, we first define two auxiliary functions for that purpose.

```
mustBeIn :: Name -> Env -> Errors
mustBeIn n e = if null (filter ((== n) . fst) e) then [n] else []

mustNotBeIn :: (Name,Int) -> Env -> Errors
mustNotBeIn p e = if p `elem` e then [fst p] else []

errors :: AGTree Errors
errors t =  case constructor t of
                       CRoot     -> errors (t.$1)
                       CNilIts   -> []
                       CConsIts  -> (errors (t.$1)) ++ (errors (t.$2))
                       CBlock    -> errors (t.$1)
                       CUse      -> (lexeme t)  `mustBeIn` (env t)
                       CDecl     -> (lexeme t,lev t) `mustNotBeIn` (dcli t)
```

Listing 3.5: Definition of *mustBeIn*, *mustNotBeIn* and *errors*

3.2.2 *Block in Kiama*

The same example presented in Haskell can also be developed in Scala via Kiama. The implementation in Kiama is very similar to the one made in Haskell, with only a few changes.

As with Haskell, first we create the syntax of the language.

```
sealed abstract class BlockTree extends Product
case class Root(its : Its) extends BlockTree

sealed abstract class Its extends BlockTree
case class ConsIts(it : It, its : Its) extends Its
case class NilIts() extends Its

sealed abstract class It extends Its
case class Decl(name : String) extends It
case class Use(name : String) extends It
case class Block(its : Its) extends It
```

Listing 3.6: Scala classes for Block

After that, we define the previously stated attributes, *dcli*, *dclo*, *env*. In Kiama we define attributes by writing the code inside an *attr* block. Another difference is that in Kiama we don't need the *lexeme* function as we can access the identifier name directly.

```
val dclo : BlockTree => Env =
        attr {
            case t@NilIts() => dcli(t)
            case ConsIts(_,its) => dclo(its)
            case t@Decl(name) => dcli(t) += ((name, lev(t))) : Env
            case t@Use(_) => dcli(t)
            case t@Block(_) => dcli(t)
            case _ => ArrayBuffer.empty
        }
```

Listing 3.7: Definition of *dclo* in Scala

```
val errors : BlockTree => Errors = {
        attr {
            case Root(its) => errors(its)
            case NilIts() => ArrayBuffer.empty
            case ConsIts(it,its) => errors(it) ++ errors(its)
            case Block(its) => errors(its)
            case t@Use(name) => mustBeIn(name, env(t))
            case t@Decl(name) => mustNotBeIn((name, lev(t)), dcli(t))
            case _ => ArrayBuffer.empty
        }
    }

    def mustBeIn(nome:String,env:Env) : Errors = {
        val e = env.filter{ case (n,_) => nome == n}
        if (e.isEmpty)
            ArrayBuffer(nome)
        else
            ArrayBuffer.empty
    }

    def mustNotBeIn(x:(String,Int),env:Env) : Errors = {
        if (env.contains(x))
            ArrayBuffer(x._1)
        else
            ArrayBuffer.empty
    }
```

Listing 3.8: Definition of *mustBeIn*, *mustNotBeIn* and *errors* in Scala

With all the other attributes defined, we create the functions *mustBeIn* and *mustNotBeIn* and finally the *errors* attribute.

3.2.3 *Benchmark*

We measured the performance of the *errors* attribute in Ztrategic (with and without memoization) and Kiama. For this, we use a function that, given an input number **n**, will output a Block value with **n** nesting levels.

The chart below shows the time in seconds of the *errors* attribute running for block sizes between 50 and 300.

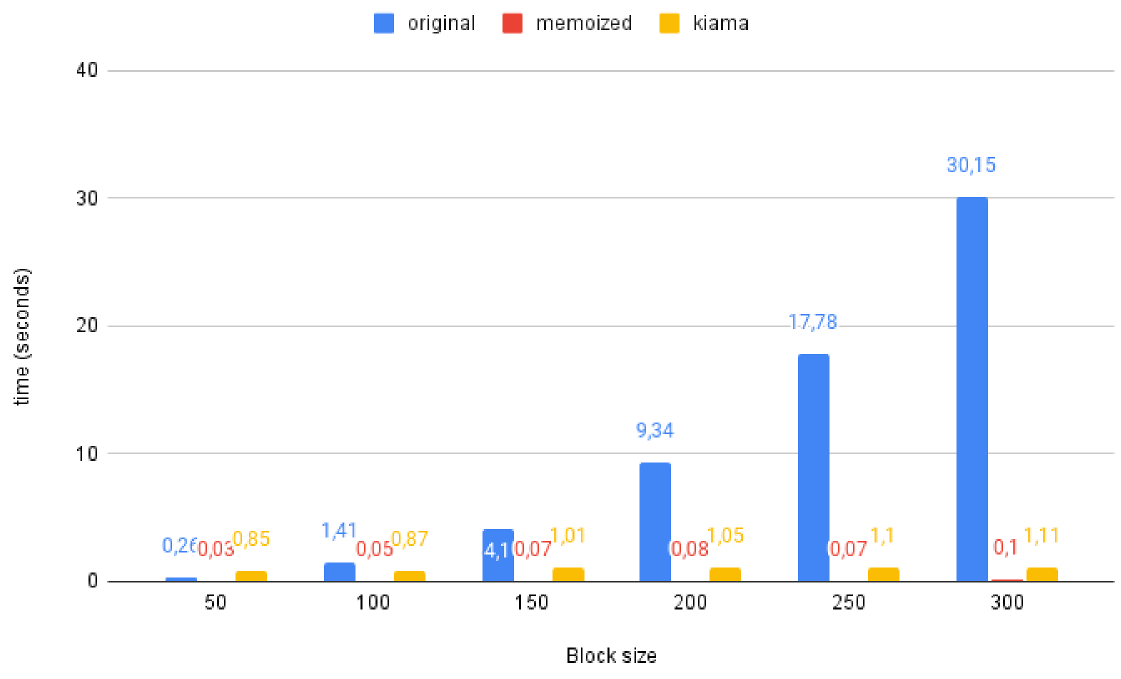

Figure 5: Non-memoized versus Memoized versus Kiama block programms

As seen in the chart, the normal version of Ztrategic is extremely slow compared to the memoized version of Ztrategic and Kiama. This happens because both Kiama and memoized Ztrategic cache its attributes.

3.3 LET

For this example, we consider the Let expressions that are present in Haskell. The semantics of Let do not require that declarations of variables occur before their first use. Likewise, a variable can only be declared once per nesting level, and only declared names can be used.

Here, we present an example of a Let expression:

```
let a = b - 16
    c = 8
    w = let  z = a + b
        in   z + b
    b = (c + 3) - c
in  c + a - w
```

Listing 3.9: Let expression

3.3.1 *Let in Ztrategic*

We begin by defining the data type for the Let expressions:

```
data Let = Let List Exp

data  List
      = NestedLet  Name Let List
      | Assign     Name Exp List
      | EmptyList

data Exp = Add   Exp Exp
         | Sub   Exp Exp
         | Neg   Exp
         | Var   Name
         | Const Int
```

Listing 3.10: Let data type

With its data types now defined, we wish to perform name analysis on our let expressions and produce a list of the variables containing errors, using a variable that was not declared, or double declaration.

The implementation of the name analysis for the Let expressions is very similar to the one done on the previous example, having the same attributes to solve the problem, but for this example we will use strategies to solve the problem.

First, we define a function that verifies that the declared variables are not in the accumulated list of declarations on the same level and another that checks if variables that are being used in the environment of its let. With these functions defined, we create a type unifying strategy that will traverse the abstract tree of the Let expression in a top-down order and apply them to its nodes.

```
type Name   = String
type Errors = [Name]

decls :: List -> Zipper Root -> [Name]
decls (Assign       s _ _) z = mNBIn (lexeme_Name z, lev z) (dcli z)
decls (NestedLet    s _ _) z = mNBIn (lexeme_Name z, lev z) (dcli z)
decls _ _                    = []

uses :: Exp -> Zipper Root -> [Name]
uses (Var i) z = mBIn (lexeme_Name z) (env z)
uses _       z = []
```

```
errors :: Zipper Root -> [Name]
errors t = applyTU (full_tdTU step) t
 where step = failTU `adhocTUZ` uses `adhocTUZ` decls

semantics :: Root -> Errors
semantics p = errors (mkAG p)
```

Listing 3.11: Name analysis startegy

With this example, we also wish to implement an arithmetic optimizer for our language. For that we define a function that will define some optimizations: the elimination of additions with 0, the sum of two Const values, replacing a subtraction with a sum, eliminate double negatives and turn a Neg (Const x) into its respective negative number (-x). Thus, we define the attribute as follows:

```
expr :: Exp -> Maybe Exp
expr (Add e (Const 0))          = Just e
expr (Add (Const 0) t)          = Just t
expr (Add (Const a) (Const b))  = Just (Const (a+b))
expr (Sub a b)                  = Just (Add a (Neg b))
expr (Neg (Neg f))              = Just f
expr (Neg (Const n))            = Just (Const (-n))
expr _                          = Nothing
```

Listing 3.12: Exp optimization

Then, we define an optimization that, given a variable x, if it's found in the environment by the env attribute, it will replace the Var x with its corresponding expression.

```
expC :: Exp -> Zipper Root -> Maybe Exp
expC (Var i) z = expand (i, lev z) (env z)
expC _ z = Nothing

expand :: (Name,Int) -> Env -> Maybe Exp
expand (i, l) e = case results of
      ((nE, lE, eE):_) -> eE
      _ -> Nothing
   where results = sortBy (\(nE1, lE1, _) (nE2, lE2, _) -> compare lE2 lE1) $
                    filter (\(nE, lE, _) -> nE == i && lE <= l) e
```

Listing 3.13: Expand optimization

These optimization rules we just implemented were first defined in (Kramer and Van Wyk, 2020), there it were defined 7 rules to the let optimization. The first 6 can be found in the listing 3.12 and the last one in listing 3.13. Because the *expC* function needs the context of the zipper in order to work, for the *lev* and *env* attributes, we separate the first 6 rules from the last. The Ztrategic library provides combinators that let us access the zipper, therefore make use of attributes like the ones referenced before with strategies.

Having defined our optimazions for Let we create the strategy that will apply them to our expressions.

```
opt'' :: Zipper Root -> Maybe (Zipper Root)
opt'' r = applyTP (innermost step) r
 where step = failTP `adhocTPZ` expC `adhocTP` expr
```

Listing 3.14: Optimization strategy

For this strategy we use an innermost traversal, this strategy will try to apply the transformation to all its nodes until it fails. As the expC function requires the zipper to work properly, we use the adhocTPZ function which will provide the zipper to our function.

3.3.2 *Let in Kiama*

As with the previous example, the attributes implement for name analysis in Let are very similar to the ones in Block, therefore next we show how to use strategies in Kiama to solve the problem.

```
  var vars = ArrayBuffer[String]()

  val uses = {
    query[Exp] {
      case t@Var(i) => vars = vars ++ mustBeIn(i, env(t))
    }
  }

  val decls = {
    query[List] {
      case t@Assign(s, _, _) => vars = vars ++ mustNotBeIn((s, lev(t)), dcli(t))
      case t@NestedLet(s, _, _) => vars = vars ++ mustNotBeIn((s, lev(t)), dcli(t
          ))
    }
  }


  def errs(e: LetTree): ArrayBuffer[String]  = {
    everywhere(decls <+ uses)(e)
    vars
  }
```

Listing 3.15: Name analysis strategy in Kiama

For this strategy, we will use a query in both Exp and List data types to obtain our error list. In Kiama, a query is combinator that always succeeds and is run to extract information without changing the tree. Then, we define our strategy, we use an everywhere traversal (top-down) and the combinator <+ to apply both functions, if decls fails then uses will be applied insted.

Next, for our optimazion strategy we start by defining the expand function, that will work the same way as in Ztrategic, and then implement the expr and expC functions to be used by the strategy.

```
  val expand : Expand => Option[Exp] = {
    attr {
      case (_, -1, _) => None
      case (n, l, e) => expandAux((n,l),e) match {
        case None => expand((n, l-1, e))
        case t => t
      }
    }
  }

  def expandAux(x : (String, Int), env : Env) : Option[Exp] = {
    if (env.isEmpty)
      None
    else
      if (x._1 == env.head._1 && x._2 == env.head._2)
        env.head._3
      else
        expandAux(x, env.tail)
  }


  val expr =
    rule[Exp] {
      case Add(e, Const(0)) => e
      case Add(Const(0), t) => t
      case Add(Const(a), Const(b)) => Const(a + b)
      case Sub(a, b) => Add(a, Neg(b))
      case Neg(Neg(f)) => f
      case Neg(Const(n)) => Const(-n)
    }

  val expC = {
    strategy[Exp] {
      case t@Var(i) => expand((i, lev(t), env(t)))
    }
  }

  val normal1 = innermost(expr <+ expC)

  def evaluate1(e: LetTree): LetTree =
    rewrite(normal1)(e)

}
```

Listing 3.16: Optimization strategy in Kiama

Our expr function will use the rule combinator for the Exp data type, this combinator will performs pattern matching on the subject term and, depending on the result of the match, return a new subject term. The expC functions uses the strategy combinator which operates similarly to rule excepet that the argument of strategy must be an Option, in this case Option[Exp]. In order to apply this strategy we use the rewrite function.

3.3.3 *Benchmark*

To compare the different versions of let, a function was defined to generate let expressions. This function was built in Haskell and Scala and takes as input the number of nesting we want, the number of nested lets for each level and a list of variables to use. Here we show the implementation in Haskell:

```haskell
testTree :: Int -> Int -> [Name] -> Root
testTree a b c = Root $ genLet a b c



genLet :: Int -> Int -> [Name] -> Let
genLet x y l = Let (genList x y l) (Add (aux x y) (genExp l))
        where aux 0 _ = Const 0
              aux _ 0 = Const 0
              aux x y = Add (Var ("nest" ++ show x ++ show y)) (aux x (y-1))



genList :: Int -> Int -> [Name] -> List
genList x y d = aux d
         where aux [] = genNest x y d
               aux (h:t) = Assign h (Const 1) (aux t)

genExp :: [Name] -> Exp
genExp [] = Const 0
genExp [h] = Var h
genExp (h:t) = Add (Var h) (genExp t)

genNest :: Int -> Int -> [Name] -> List
genNest 0 _ _ = EmptyList
genNest _ 0 _ = EmptyList
genNest x y d = NestedLet ("nest" ++ show x ++ show y) (Let (genList (x-1) y d) (
    Add (aux x y) (genExp d))) (aux2 (y-1))
        where aux 1 _ = Const 0
              aux _ 0 = Const 0
              aux x y = Add (Var ("nest" ++ show (x - 1) ++ show y)) (aux x (y-1)
                  )
              aux2 0 = EmptyList
              aux2 yy = NestedLet ("nest" ++ show x ++ show yy) (Let (genList (x
                  -1) y d) (Add (aux x y) (genExp d)) ) (aux2 (yy-1))
```

Listing 3.17: Let generator

If we call this function with 1 nest, 2 nested lets per level and with variables a and b, we will obtain a let like this:

```
let a = 1
    b = 1
    nest12 = let a = 1
                 b = 1
             in 0 + a + b
    nest11 = let a = 1
                 b = 1
             in 0 + a + b
in nest12 + nest11 + 0 + a + b
```

Listing 3.18: Let generator example

In our benchmarks we always use 2 nests and the variables a,b and c, only changing the number of nested lets per level. This graphs show us the performance of name analysis(name), optimization(opt), and both together(name - opt) for the normal version of Ztrategic, memo Ztrategic and Kiama, using 5, 10 and 20 nested lets per level.

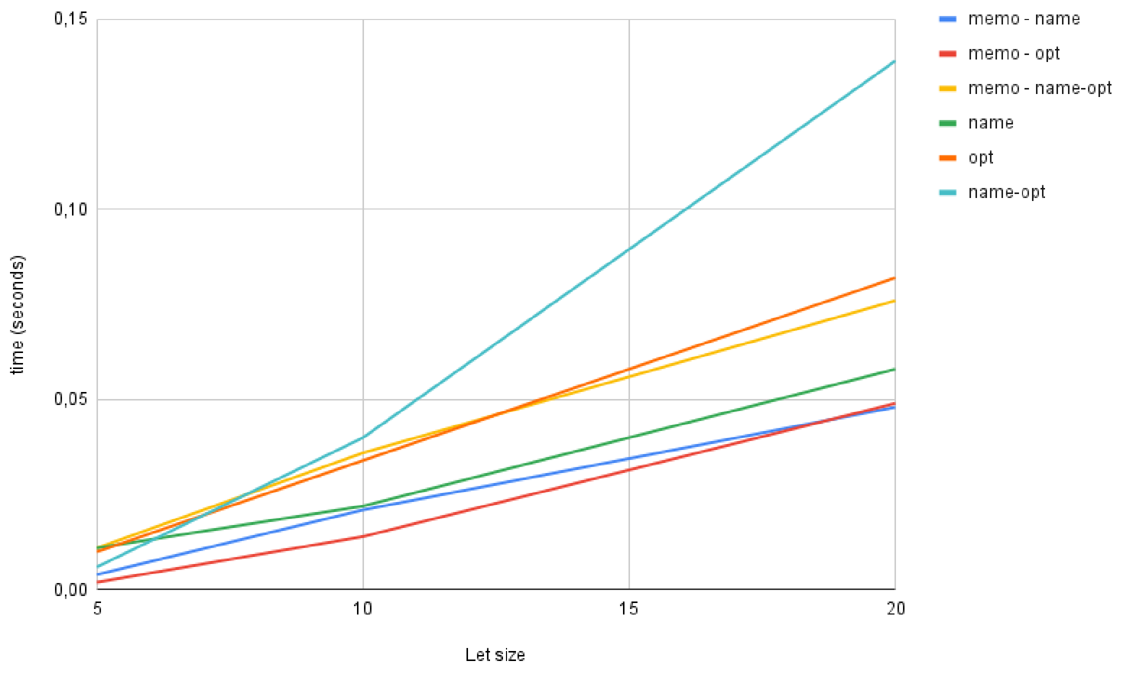

Figure 6: Non-memoized versus Memoized let programms

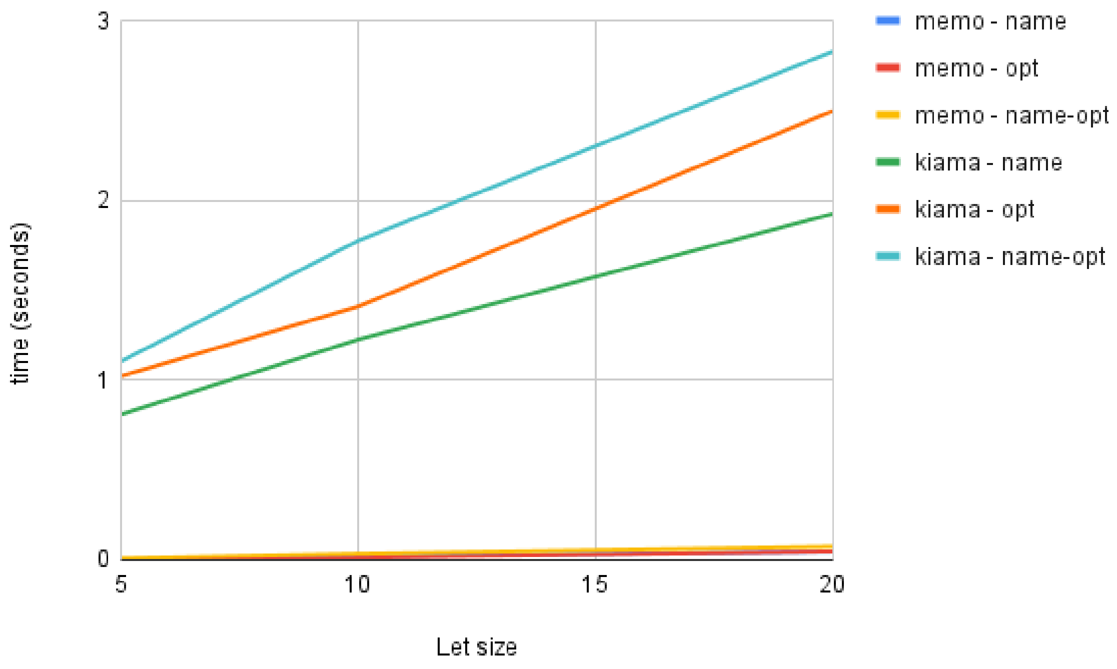

Figure 7: Memoized versus Kiama let programms

Looking at figures 6 and 7 we see that the kiama implemention is the slowest one. As we are using relatively small let examples, the attribute caching Kiama does is not being usefull to these expressions. If we use bigger let examples, we expect that kiama will be faster than our normal, non memoized version. Between our Ztrategic implementations, as expected, the memoized version has better performance than our normal one.

3.4 PRETTY PRINTING

The pretty printing algorithm that we present in this section was first defined and implemented in Swierstra et al. (1999). This is a very big and complex algorithm that allows us to print our texts in various ways, even allowing multiple layouts, that is, we can define different ways to print our text and the algorithm will pick the best one based on the space given to print it.

Some of the basic combinators for this algorithm are:

- text - converts a string to the data type used by the algorithm
- indent - indents the text by a given number
- empty - no text
- (>|<) - places two arguments besides each other
- (>-<) - places two arguments above each other

- (>#<) - places two arguments besides each other, with a space inbetween

To print our text, we use the function *render* that given the text and the page size, renders the text to the screen. For example, if we wanted to print "Hello World!" using our pretty printing algorithm, we would do so in the following way.

```
x = (text "Hello") >#< (text "World!")

render x 10
```

Listing 3.19: Pretty printing example

In listing 3.19, we use the *text* function to convert our text to the data type used by the algorithm and then join the text using *>#<* for it to render side by side with a space inbetween. The render function will take our text and the size of where it will print, in this case the size given was 10.

We implemented our pretty printing in both Ztrategic and Kiama and then, using the problem in section 3.3 we tried to print some let expressions with both our implementations.

Using the following let example, we measured the performance of our different versions.

```
let a = b + -16
    c = 8
    w = let z = a + b
        in z + b
    b = c + 3 - c
    t = let a = b + -16
            c = 8
            w = let z = a + b
                in z + b
            b = c + 3 - c
        in c + a - w
    t = let a = b + -16
            c = 8
            w = let z = a + b
                in z + b
            b = c + 3 - c
        in c + a - w
    t = let a = b + -16
            c = 8
            w = let z = a + b
                in z + b
            b = c + 3 - c
        in c + a - w
    t = let a = b + -16
            c = 8
            w = let z = a + b
                in z + b
            b = c + 3 - c
```

```
         in c + a − w
in c + a − w
```

Listing 3.20: Pretty print let example

With Ztrategic, we could print this let in less than a second, more precisely, in 0,03 seconds. On the other hand, with Kiama, we got times of around 102 seconds (1,7 minutes).

4 CONCLUSIONS

In this thesis, we have studied the combination of strategic programming with attribute grammar programming. We implemented several language engineering algorithms using both AGs and strategies. Furthermore, we have benchmarked the Ztrategic library and compared it with the state of the art Kiama library. The results of this study allow us to answer our original research questions.

- *Research Question 1:* Is the combined memoized embedding of Strategic AGs expressive enough to specify complex language engineering algorithms?

  Answer: We have expressed in the combined embedding of strategic term re-writing and AGs a large and complex algorithm: the optimal pretty printing algorithm presented in Swierstra et al. (1999). This is a non-trivial four traversal algorithm which was expressed in our setting in the elegant AG programming style. Moreoever, we also expressed this algorithm in the Kiama setting.

- *Research Question 2:* How does the performance of memoized strategic AGs compare to the state-of-the-art defined by Kiama (Sloane et al., 2010)?

  Answer: We benchmarked both our and Kiama's when specifying well-known language processing tasks. Our results show that our memoized embedding of strategies and AGs is already faster than the state-of-the-art Kiama system. Moreover, when considering the four traversal optimal pretty printing algorithm, the Kiama solution is very inefficient taking too long to pretty print a small example.

BIBLIOGRAPHY

Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In Franz Baader, editor, *Term Rewriting and Applications*, pages 36–47, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73449-9.

James R. Cordy. Txl - a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2004.11.006.

Oege de Moor, Kevin Backhouse, and Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000. URL citeseer.ist.psu.edu/demoor00firstclass.html.

Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an attribute grammar. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, pages 1–72, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31872-9.

Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, October 2007. ISSN 0362-1340. URL http://doi.acm.org/10.1145/1297105.1297029.

João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, and Marcos Viera. Memoized zipper-based attribute grammars and their higher order extension. *Sci. Comput. Program.*, 173:71–94, 2019. doi: 10.1016/j.scico.2018.10.006. URL https://doi.org/10.1016/j.scico.2018.10.006.

Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, February 1992. ISSN 0001-0782. URL https://doi.org/10.1145/129630.129637.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. ISSN 0956-7968.

Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 1968.

Lucas Kramer and Eric Van Wyk. Strategic tree rewriting in attribute grammars. pages 210–229, 11 2020. doi: 10.1145/3426425.3426943.

Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, pages 137–154, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45587-5.

Sebastiaan P. Luttik and Eelco Visser. Specification of rewriting strategies. In *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications*, Algebraic'97, page 9, Swindon, GBR, 1997. BCS Learning & Development Ltd. ISBN 3540762280.

Ralf Lämmel and Joost Visser. A strafunski application letter. pages 357–375, 01 2003. ISBN 978-3-540-00389-2. doi: 10.1007/3-540-36388-2_24.

José Nuno Macedo, Emanuel Rodrigues, Marcos Viera, and João Saraiva. Efficient embedding of strategic and attribute grammars via memoization and zippers. (submitted), 2022a.

José Nuno Macedo, Marcos Viera, and João Saraiva. Zipping strategies and attribute grammars. In Michael Hanus and Atsushi Igarashi, editors, *Functional and Logic Programming*, pages 112–132, Cham, 2022b. Springer International Publishing. ISBN 978-3-030-99461-7.

José Nuno Macedo, Marcos Viera, and João Saraiva. Zipping strategies and attribute grammars, 2021.

Pedro Martins, João Fernandes, and João Saraiva. Zipper-based attribute grammars and their extensions. pages 135–149, 10 2013. ISBN 978-3-642-40921-9. doi: 10.1007/978-3-642-40922-6_10.

Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. Lisa: A tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79, April 1995. ISSN 0362-1340. URL https://doi.org/10.1145/202176.202185.

Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48, April 1984. ISSN 0362-1340. URL http://doi.acm.org/10.1145/390011.808247.

Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. *Electronic Notes in Theoretical Computer Science*, 253(7):205–219, 2010. ISSN 1571-0661. URL http://dx.doi.org/10.1016/j.entcs.2010.08.043.

S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, pages 150–206, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. URL https://doi.org/10.1007/10704973_4.

Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, page 365–370, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 354041861X.

A. van Wijngaarcien, B. J. Mailloux, J. E. L. Peck, C. H. A. Kostcr, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised Report on the Algorithmic Language Algol 68. *SIGPLAN Not.*, 12(5):1–70, May 1977. ISSN 0362-1340. URL https://doi.org/10.1145/954652.1781176.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008. ISSN 1571-0661. URL http://dx.doi.org/10.1016/j.entcs.2008.03.047.

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, RTA '01, page 357–362, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3540421173.