

pyZtrategic: A Zipper-Based Embedding of Strategies and Attribute Grammars in Python

Emanuel Rodrigues¹^a, José Nuno Macedo¹^b, Marcos Viera²^c and João Saraiva¹^d

¹HASLab & INESC TEC, University of Minho, Braga, Portugal

²Universidad de la República, Montevideo, Uruguay

fi

Keywords: Attribute Grammars, Strategic Term Rewriting, Software Analysis and Evolution.

Abstract: This paper presents pyZtrategic: a library that embeds strategic term rewriting and attribute grammars in the Python programming language. Strategic term rewriting and attribute grammars are two powerful programming techniques widely used in language engineering: The former relies on strategies to apply term rewrite rules in defining large-scale language transformations, while the latter is suitable to express context-dependent language processing algorithms. Thus, pyZtrategic offers Python programmers recursion schemes (strategies) which apply term rewrite rules in defining large scale language transformations. It also offers attribute grammars to express context-dependent language processing algorithms. PyZtrategic offers the best of those two worlds, thus providing powerful abstractions to express software maintenance and evolution tasks. Moreover, we developed several language engineering problems in pyZtrategic, and we compare it to well established strategic programming and attribute grammar systems. Our preliminary results show that our library offers similar expressiveness as such systems, but, unfortunately, it does suffer from the current poor runtime performance of the Python language.


1 INTRODUCTION


Modern software languages offer powerful mechanisms to improve the productivity of their programmers, like powerful type and modular systems, bad smell detection and refactorings, etc. To design and implement such mechanisms, we need powerful techniques to analyse, manipulate and evolve such languages.


Strategic term rewriting (Luttik and Visser, 1997; Visser et al., 1998) and Attribute Grammars (AG) (Knuth, 1968) have a long history in supporting the development of modern software language analysis, maintenance, refactoring, evolution and optimizations. The former relies on recursion schemes, called *strategies*, to traverse a tree while applying a set of rewrite rules, while the latter is suitable to express static context-dependent language processing algorithms. The expressiveness and usefulness of such techniques can be seen by the many language systems


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). In fact, these systems have been used to express software maintenance and evolution of large (real) language/programs, such as smell detection and program refactoring (Lämmel and Visser, 2002; Macedo et al., 2024).

Most of these powerful systems, however, are large language engineering systems that support their own AG or strategic notation. These systems require a considerable development effort to build, to maintain, and to extend. As a result, most of these systems support one of the techniques only. A more flexible approach is obtained when we consider the embedding of such techniques in a general purpose language. A language embedding, however, usually relies on advanced mechanisms of the host language. In this paper, we present the pyZtrategic system which embeds both strategic term rewriting and attribute grammars in the Python language. PyZtrategic offers the first embedding of these techniques in Python, thus making such powerful

^a <https://orcid.org/0000-0003-4317-1144>

^b <https://orcid.org/0000-0002-0282-5060>

^c <https://orcid.org/0000-0003-2291-6151>

^d <https://orcid.org/0000-0002-5686-7151>

language engineering techniques available to Python programmers.

In this paper, we show how our Python strategic term rewriting embedding can be used to express source code optimisations. Moreover, we use AGs to express scope rules in the same language and show how it can be combined with strategies to provide powerful context dependent tree transformations. Finally, we compare the expressiveness of pyZstrategic with the well-known Stratego system (Visser, 2001) and we benchmark our library against the Zstrategic library (Macedo et al., 2022): a Haskell combined embedding of strategies and AGs. Our first results show that the expressiveness of our library is similar to Stratego strategic notation. Python is known to be a slow language (Pereira et al., 2021) and our benchmarks show this poor performance of the language: pyZstrategic is slower than the similar Zstrategic Haskell library (Macedo et al., 2022; Macedo et al., 2024).

This paper is organised as follows: Section 2 presents a usage example using Stratego and then our zipper-based embedding of strategic term rewriting. In Section 3, we show how to combine zipper-based strategic term rewriting with attribute grammars. In Section 4 we compare our library against Stratego and Zstrategic and use it to define usage examples. Section 5 discusses related work, and in Section 6 we present our conclusions.

2 ZIPPER-BASED STRATEGIC PROGRAMMING

Both strategic term-rewriting (Luttik and Visser, 1997; Lämmel and Visser, 2002) and attribute grammars (Knuth, 1968) rely on generic (abstract syntax) tree traversal mechanisms to navigate on trees (Saraiva and Swierstra, 1999). The former to express large scale tree transformations (Visser et al., 1998; Lämmel and Visser, 2003) while the latter to express context-dependent algorithms (Saraiva, 1999; Saraiva, 2002). Because most useful large scale transformations rely on context information, there are several approaches that combine both techniques, namely the Kiama framework (Sloane et al., 2010) - an embedding of strategies and AGs in the Scala language - and the Zstrategic library (Macedo et al., 2022) - a Haskell-based embedding of both techniques.

This paper presents the pyZstrategic framework: a multi paradigm embedding of strategies and AGs in the Python programming language. This section discusses the embedding of strategies in Python, while in Section 3.1 we combine this embedding with AGs.

Before presenting this embedding in detail, let us

$$add(e, const(0)) \rightarrow e \quad (1)$$

$$add(const(0), e) \rightarrow e \quad (2)$$

$$add(const(a), const(b)) \rightarrow const(a + b) \quad (3)$$

$$sub(e1, e2) \rightarrow add(e1, neg(e2)) \quad (4)$$

$$neg(neg(e)) \rightarrow e \quad (5)$$

$$neg(const(a)) \rightarrow const(-a) \quad (6)$$

$$var(id) \mid (id, just(e)) \in env \rightarrow e \quad (7)$$

Figure 1: Optimization Rules.

consider a motivating example we will use throughout the paper. Consider the (sub)language of *Let* expressions, as usually expressed in functional languages. Next, we show an example of a valid *let* expression.

```
p = let  a = b + 0
      c = 2
      b = let c = 3 in c + c
      in  a + 7 - c
```

In the definition of p several names are defined in the same scope. Typically, in *let* expressions, defining the same name twice in the same scope is a semantic error, as well as using a non-defined name. It is also valid to re-define names that were defined in an outer scope. For example, in p the name c is re-defined in a nested *let* expression.

Our objective with this example is to define an optimizer for such *let* expressions. This optimizer follows the rules presented in Figure 1 given in (Kramer and Van Wyk, 2020). Rules 1 through 6 are relatively simple, as they consist of matching certain patterns in an expression and replacing them. For example, rules 1 and 2 define that the addition of any expression e with the constant value 0 should be replaced by just the expression e . In fact, rules 1 to 6 are the perfect setting to be expressed as a strategic program, as we will show in the next subsection.

Rule 7, however, requires the knowledge of the environment env , which contains all defined names in the scope and its respective values. The rule itself defines that any variable id can be replaced by its value as defined in the environment. Strategic programming, however, is context-free and, consequently, it does not provide a natural setting to express such rules. In Section 3, we use the same Python zipper-based setting to embed attribute grammars: a suitable formalism to define context-dependent computations. By using a uniform setting to express both the strategic and AGs embeddings, we can easily combine the two formalisms and get the best of both worlds. As a result, our multi-paradigm embedding provides an elegant setting to express context-dependent re-writings such as required by rule 7.

2.1 Stratego

In order to present strategic term-rewriting, we start by presenting a strategic program expressed in the notation used by the well-known Stratego system. The Stratego system, developed by Eelco Visser, is a widely used and powerful strategic term rewriting based system which is part of the Spoofox Language Designer's Workbench (Kats and Visser, 2010). Stratego uses a domain specific language (DSL) to express algebraic data types, re-writing rules, and strategies to apply such rules. Stratego is based on a traditional architecture: it includes a language processor that translates the Stratego specification into an efficient strategic program (expressed in C). Such systems, however, tend to be large, complex, and thus difficult to maintain and evolve. In this section, we show an embedding of strategic term re-writing that does not rely on such a large language system. Instead, it uses techniques to embed a DSL - defining a strategic program - directly into the Python general purpose language.

In order to briefly introduce strategic term re-writing, we start by expressing our running example in Stratego (rules 1 to 6). At the end of this section, we will present an equivalent and very similar solution, but now fully expressed as an embedded DSL in Python.

Let us start by defining the abstract syntax of the let (sub)language via Stratego abstract data type notation.

```
signature
  sorts
    Let List Exp

  constructors
    Let          : List * Exp -> Let
    NestedLet   : STRING * Let * Exp
    ↪ -> List
    Assign      : STRING * Let * Exp
    ↪ -> List
    Empty       : List
    Add         : Exp * Exp -> Exp
    Sub        : Exp * Exp -> Exp
    Neg        : Exp -> Exp
    Var        : STRING -> Exp
    Const      : INT -> Exp
```

We omit here the explanation of these data types, since they are self-explanatory.¹ Let us consider now that we wish to implement the simple arithmetic optimizer for our example. In Stratego we define the first 6 rules shown in Figure 1 as follows.

¹The reader can also find ample documentation about the Stratego system from its webpage.

```
rules
  Opt : Add(Const(0), x) -> x
  Opt : Add(x, Const(0)) -> x
  Opt : Add(Const(x), Const(y)) ->
    ↪ Const(<add> (x, y))
  Opt : Sub(x, y) -> Add(x, Neg(y))
  Opt : Neg(Neg(x)) -> x
  Opt : Neg(Const(x)) -> Const(<mul> (x
    ↪ , -1))
```

These 6 rules work on a single node of the tree. In order to express the application of this re-writing while traversing the tree, we need to use pre-defined Stratego strategies. Next, we show the strategic solution of our optimization where *expr* is applied to the input tree in an innermost strategy. This strategy performs a transformation as many times as possible, starting from the inside nodes.

```
strategies
  main = io-wrap(eval)
  eval = innermost(Opt)
```

As mentioned before, strategic term rewriting does not provide a natural way to express rule 7. The Stratego system is no exception. In Section 3 we will provide an elegant solution for rule 7. Next, we show how to embed strategic term rewriting in Python and to express rule 1 to 6 directly as a Python program.

2.2 Python Embedding of Strategies

The Stratego system uses standard language engineering techniques to implement a DSL. Indeed, it implements a language processor for that DSL. PyZtrategic uses a different approach: it embeds strategies in the Python programming language. The idea is that writing a strategic program in pyZtrategic is similar to express it in Stratego. The key advantage of pyZtrategic is that we do not have to implement a language processor from scratch for the strategic DSL.

Next, we show the key ingredients of our embedding, namely the use of algebraic data types, the definition of type (node) specific transformations by relying on pattern matching, and the use of our Python strategic combinator library to perform such specific transformation while traversing the tree using a reusable recursion pattern (*i.e.* strategy).

One of the key ingredients of most strategic term rewriting systems (Kats and Visser, 2010; Lämmel and Visser, 2002; Macedo et al., 2022; Sloane et al., 2010) is the use of algebraic data types to express the abstract syntax of the language under analysis/transformation. Algebraic data types are not native in Python. Thus, we rely on a Python library² that offers algebraic data

²<https://pypi.org/project/algebraic-data-types/>

types to Python programmers. Using this library, we are able to express the previous Stratego definition of the let language directly in Python.

```
@adt
class Let:
    LET: Case["List", "Exp"]

@adt
class List:
    NESTEDLET: Case[str, "Let", "List"]
    ASSIGN: Case[str, "Exp", "List"]
    EMPTY: Case

@adt
class Exp:
    ADD: Case["Exp", "Exp"]
    SUB: Case["Exp", "Exp"]
    NEG: Case["Exp"]
    VAR: Case[str]
    CONST: Case[int]
```

Each data type (or grammar symbol) corresponds in Python to a class, upon which we apply the `@adt` decorator. For each constructor/production, we declare a field with its case annotation. As a result, we are able to express the abstract syntax very much like Stratego programmers do.

Having defined the algebraic data type, we can write p directly as a Python expression:

```
p = Let.LET(List.ASSIGN("a",
→ Exp.ADD(Exp.VAR("b"), Exp.CONST(0)),
    List.ASSIGN("c", Exp.CONST(2),
    List.NESTEDLET("b",
→ Let.LET(List.ASSIGN("c",
→ Exp.CONST(3), List.EMPTY()),
→ Exp.ADD(Exp.VAR("c"),
→ Exp.VAR("c"))),
    List.EMPTY()))),
Exp.SUB(Exp.ADD(Exp.VAR("a"),
→ Exp.CONST(7)), Exp.VAR("c")))
```

The algebraic data type library also offers pattern matching that we may use to define a type/node specific transformation, in the form of the `match` function. Now we define the first 6 rules in Python as follows:

```
def optAdd(x, y):
    # rule 1
    if (y == Exp.CONST(0)):
        return x
    # rule 2
    elif (x == Exp.CONST(0)):
        return y
    # rule 3
    elif (lambda a, b: x == Exp.CONST() and
→ y == Exp.CONST()):
        return Exp.CONST(x.const() +
→ y.const())
    else:
        return st.StrategicError
```

```
def optNeg(x):
    # rule 5
    if (lambda a: x == Exp.NEG()):
        return x.neg()
    # rule 6
    elif (lambda b: x == Exp.CONST()):
        return Exp.CONST(-x.neg())
    else:
        return st.StrategicError

def expr(exp):
    x = exp.match(
        add=lambda x, y: optAdd(x, y),
        neg=lambda x: optNeg(x),
        var=lambda x: st.StrategicError,
        const=lambda x: st.StrategicError,
        #rule4
        sub=lambda x, y: Exp.ADD(x,
→ Exp.NEG(y))
    )
    if x is st.StrategicError:
        raise x
    else:
        return x
```

Functions `optAdd` and `optNeg` check for specific patterns and apply the optimizations if said patterns are found. We include comments in the code for better readability on where each rule is defined in these functions. In function `expr`, we check what the constructor of the node is, and apply the corresponding optimizations for that constructor. For `Add` nodes, we call `optAdd`, and so on.

As the `match` function requires the pattern matching to be total, we have to return a result for constructors which we do not want to modify; for this, we return the `StrategicError` value. This is an exception defined in the `pyZstrategic` library to signal failure, and thus we signal it by raising the exception when we are not interested in modifying a node. Because `StrategicError` is an exception, we need an `if` clause to disambiguate if the computed value x is an exception to be raised, or a value to be returned.

As we mentioned, `expr` works on a single tree node, only. In order to apply this rule to all possible nodes in the tree, we need a generic tree traversal mechanism like the one offered by Stratego strategic combinators (for example, function `innermost`).

Having expressed all rewriting rules from 1 to 6 in `expr`, now we need to use our strategic combinators that navigate in the tree while applying the rules. These combinators rely on the Zipper data structure: a generic mechanism to navigate on heterogeneous trees (Huet, 1997)³. To guarantee that all the possible optimizations are applied, we use an `innermost`

³Zipperers are also implemented as a Python library <https://pypi.org/project/zipper/>

traversal scheme. Thus, our optimization is expressed as:

```
def optR(z):
    return st.innermost(lambda x:
        → st.adhocTP(st.failTP, expr, x),
        → z).node()
```

The transformation *failTP* is an always failing transformation (raises our *StrategicError* exception) and the *adhocTP* combinator joins two transformations into a single one by trying to apply the rightmost function, and if it fails, applies the left one. Therefore, the result of the usage of *adhocTP* here is a transformation function that attempts to apply *expr* when possible, and in any other cases, it fails due to *failTP*. The *innermost* traversal scheme will attempt to apply this transformation as many times as possible until a fixed point is reached.

Rule 7 is context dependent and requires first to compute the environment where a name is used. Rewriting rules relying on context information are not easily expressed as pure strategic definitions. The computation and flow of context information, in a (abstract syntax) tree, is the natural setting for AGs. Thus, we will return to rule 7 after we extend our Python embedding with AGs.

3 STRATEGIC ATTRIBUTE GRAMMARS

There are transformations that rely on contextual information that needs to be collected first so that the transformation can be applied. Rule 7 from Figure 1 is such a case. This section will explain how to combine strategies with AGs, showing how to implement rule 7.

3.1 Zipper-Based Attribute Grammars

We show a visual representation of our AG in Fig.2 along with its definition in Python. In Fig.2, productions are shown with the parent node above and children nodes below, inherited attributes are on their left and synthesized attributes on their right, and arrows show how information flows between productions and their children to compute attributes.

In this AG, the attribute *dcli* is an accumulator, used to collect all variables defined in a *let*. The complete list is synthesized in the attribute *dclo*. Our attributes will be a list of triples containing the variable name, the level where it's defined (to distinguish between declarations of the same name) and the expression associated with it.

We start with the synthesized attribute *dclo*. For example, looking at the diagram, we verify that the

dclo of Let productions is the *dclo* of its first child. On the other hand, on the *EmptyList* production, the *dclo* is a copy of *dcli* (presented later in this section).

```
def dclo(x):
    match constructor(x):
        case Constructor.CRoot:
            return dclo(x.z_dollar(1))
        case Constructor.CLet:
            return dclo(x.z_dollar(1))
        case Constructor.CNestedLet:
            return dclo(x.z_dollar(3))
        case Constructor.CAssign:
            return dclo(x.z_dollar(3))
        case Constructor.CEmpty:
            return dcli(x)
```

The attribute *lev* is used to distinguish declarations with the same name in different levels. It is omitted in our diagram due to its simple nature. This attribute is passed downwards as a copy of the parent node, with two exceptions: when in a Let subtree whose parent is a Root, and when visiting a *NestedLet*. In the former the level is 0, while in the latter, since we are descending to a nested block, we increment the level of the outer one.

```
def lev(x):
    match constructor(x):
        case Constructor.CLet:
            match constructor(x.up()):
                case Constructor.CNestedLet:
                    return lev(x.up()) + 1
                case Constructor.CRoot:
                    return 0
        case _:
            return lev(x.up())
```

Now, let us consider the accumulator attribute *dcli*. The function, when visiting nodes of type Let, has to consider two alternatives: the parent node can be a Root or a *NestedLet*. This happens because the rules to define its value differ: in the Root node it corresponds to an empty list, while in a nested block, the accumulator *dcli* starts as the env of the outer block. Finally, there are three different cases: when the parent is a Let node, *dcli* is a copy of the parent. When the parent is an Assign, then the Name, level and the associated Exp are accumulated in the *dcli* of the parent. Finally, in the case of *NestedLet* the Name, level, and an exception are accumulated in *dcli*.

```
def dcli(x):
    match constructor(x):
        case Constructor.CLet:
            match constructor(x.up()):
                case Constructor.CRoot:
                    return []
                case Constructor.CNestedLet:
                    return env(x.up())
        case _:
            match constructor(x.up()):
```

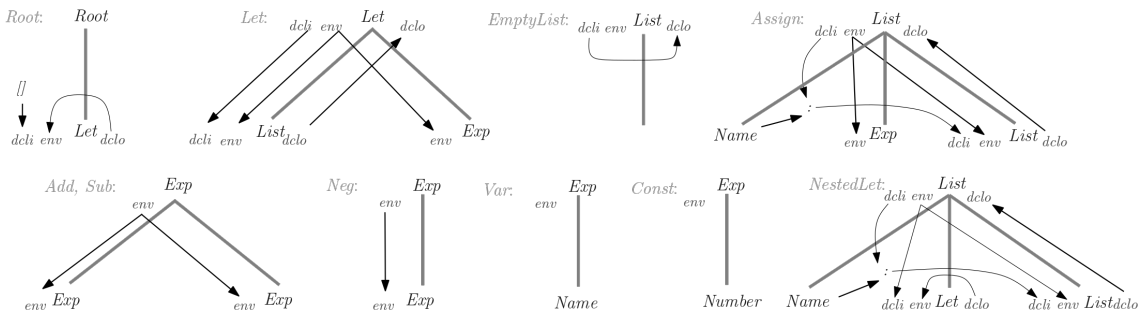


Figure 2: Attribute Grammar Specifying the Scope Rules of Let.

```

case Constructor.CLet:
    return dcli(x.up())
case Constructor.CAssign:
    return
    ↪ [(lexeme_Name(x.up()),
    ↪ lev(x.up()),
    ↪ lexeme_Exp(x.up()))] +
    ↪ dcli(x.up())
case Constructor.CNestedLet:
    return
    ↪ [(lexeme_Name(x.up()),
    ↪ lev(x.up()),
    ↪ st.StrategicError)] +
    ↪ dcli(x.up())
    
```

```

def expC(exp, z):
    x = exp.match(
        add=lambda x, y: st.StrategicError,
        sub=lambda x, y: st.StrategicError,
        neg=lambda x: st.StrategicError,
        var=lambda x: expand((x, lev(z)),
            ↪ env(z)),
        const=lambda x: st.StrategicError
    )
    if x is st.StrategicError:
        raise x
    else:
        return x
    
```

Finally, we have the *env* attribute. In most diagrams, an occurrence of attribute *env* is defined as a copy of the parent. There are two exceptions: in productions *Root* and *NestedLet* where *Let* subtrees occur. In both cases, *env* gets its value from the synthesized attribute *dcli* of the same non-terminal/type. We use the default rule of the case statement to express similar AG copy equations.

```

def env(x):
    match constructor(x):
        case Constructor.CRoot:
            return dcli(x)
        case Constructor.CLet:
            return dcli(x)
        case _:
            return env(x.up())
    
```

3.2 Strategic Attribute Grammars

As we mention before, rule 7 requires knowledge of the context to be implemented. This rule states that an occurrence of a variable can be changed by its definition. Therefore, we will need an environment of all the defined variables, which we have done with the *env* attribute. In order for the strategy to expose the zipper so that it can be used to compute a given attribute, we will make use of a new *ad hoc* function called *ad hocTPZ*.

Then, we can define a function that implements our rule 7:

Note that *expC* has two arguments, *exp* which is the current node to be processed, and *z* which is the zipper pointing to our current position. With *z* now visible, we can compute the attributes *lev* and *env* using it. We use an auxiliary *expand* function that looks up the variable *x* in the environment produced by *env*, with a nesting level not bigger than the value computed by *lev*.

Finally, we combine rule 7 with the others previously defined. For readability, we split rule 7 into a definition named *exp1*, and the combination of *exp1* with the first 6 rules in *exp2*. We apply *exp2* as many times as possible using the *innermost* strategy, and the resulting function is named *optRC*.

```

def optRC(z):
    def exp1(y):
        return st.adhocTPZ(st.failTP, expC,
            ↪ y)

    def exp2(x):
        return st.adhocTP(exp1, expr, x)

    return st.innermost(exp2, z).node()
    
```

4 EXPRESSIVENESS AND PERFORMANCE

In this section, we compare the expressiveness of the pyZstrategic Python library with the well-known and established Stratego system. After that, we benchmark

our embedding, and we compare its runtime performance with the Zstrategic library⁴: a Haskell based embedding of both techniques (Macedo et al., 2022).

4.1 pyZstrategic versus Stratego

We have presented the Stratego and pyZstrategic solution for our let optimization problem. The two solutions are similar. However, because Stratego uses a proper notation to express re-writing rules, it offers more concise and readable solutions than our embedding. This can be clearly seen in the definition of the type-specific transformation where the lack of a pre-defined pattern matching mechanism in Python, makes the solutions longer and harder to read. This is, however, a well-known disadvantage of using a pure embedded DSL approach when compared to the use of proper notation that is processed by (a large and complex) language processor. The other disadvantage of using our Python embedding concerns errors reporting: while the Stratego processor can produce error messages in the context of a strategic program, our Python embedding relies on the (dynamic and more liberal) type system of Python and, thus, errors are reported as general Python errors.

There is, however, a key advantage of pyZstrategic: it is a simple library that does not require a large/complex language processor. A language processor is always a complex and time-consuming system to build, maintain and evolve. An embedded DSL, such as the one defined by pyZstrategic, uses the language infrastructure of the host language and can easily be maintained - by using Python tools such as debuggers, profiles, etc - and evolved - which consists in defining new combinators to the existing Python library.

4.2 pyZstrategic versus Zstrategic

The Zstrategic library in the Haskell programming language presents a similar embedding of zippers and attribute grammars as our Python solution. However, there are some key differences between the two:

- While in Python, we need to use a library to provide support for algebraic data types, Haskell provides a way to create them by default.
- The Python ADT library offers a poor pattern matching mechanism when compared to Haskell, where we can have more elegant worker functions.
- The Python programming language offers generally poor performance when compared to Haskell, and performance differences are also noticeable when using both libraries.

⁴<https://bitbucket.org/zenunomacedo/zstrategic/>

We ran an advanced Software Maintenance and Evolution course for Master's Degree students, for which the students had to develop a parser, optimizer and pretty-printer for the Unix BC calculator language. For the optimizer, the students had to use strategic programming, with freedom to choose either Python pyZstrategic or Haskell Zstrategic libraries since they were familiar with both programming languages. Next, we select two solutions developed by our students using both libraries, and we compare their number of lines of code. We believe these solutions to be representative of the average usage experience for these libraries. The results are shown in Table 1.

Note that these solutions are not strictly similar. For example, the Python solution uses the Lark parsing toolkit to build a parser, while the Haskell solution uses parser combinators. The Haskell solution implements 52 optimizations, while the Python solution implements 29 optimizations. Nevertheless, as expected, these results illustrate that Python implementations tend to be more verbose when compared to their Haskell counterpart.

Table 1: Number of Lines of Code - Python vs Haskell.

	Python	Haskell
Data Type Declarations	219	32
Parser	90	78
Pretty-Printer	49	42
Strategy	6	3
Optimizations	208	76

Next, we will compare the performance of the pyZstrategic library with Zstrategic and Kiama, a Scala lightweight language processing library for attribute grammars and strategy-based term rewriting.

4.3 Let Optimization

We implemented the let optimizer presented in this paper in both pyZstrategic, Zstrategic and Kiama, as all libraries provide Attribute Grammars and strategies. In Figure 3 we show the performance of optimizing several Let inputs of different sizes in terms of runtime and memory usage. Let size, as seen in the X axis of the figure, refers to the number of nested let blocks present in the input. As we can see in the figure, our pyZstrategic implementation presents the poorest runtime performance of the three. We expect bad runtime performance for Python implementations when compared to other, more efficient languages, which both Haskell and Scala qualify as.

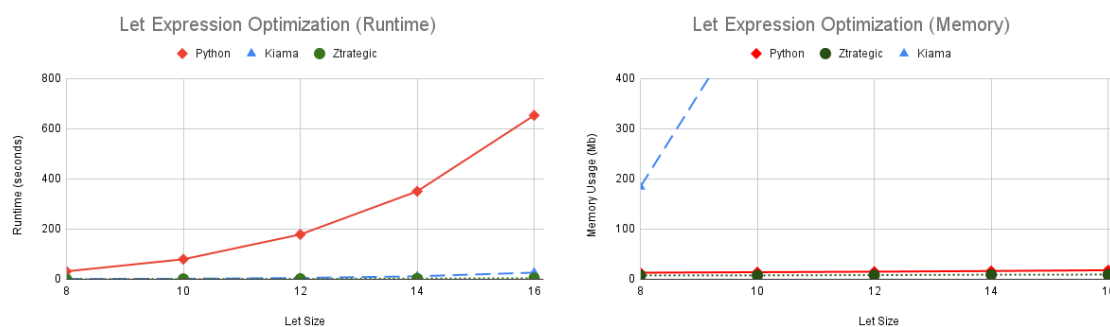


Figure 3: Let Optimization: Python versus Zstrategic versus Kiama.

4.4 Reprim

The *reprim* problem is a well-known one. The goal of it is to transform a binary tree of integers into a new one with the same shape, but where all leaves are replaced by the minimum leaf value of the original tree. We solve the *reprim* problem using a strategy to compute the minimum value of the tree, and then another strategy to propagate the computed value throughout the tree.

In Figure 4 we show our implementation results. The *Reprim size* refers to the number of nodes the input tree contains. Again, Zstrategic outperforms both pyZstrategic and Kiama, who behave similarly in terms of runtime. The memory consumption is similar, except for Kiama, who presents by far the poorest performance.

5 RELATED WORK

The Zstrategic (Macedo et al., 2022; Macedo et al., 2024) library is an embedded library for combining Attribute Grammars and strategic programming in the Haskell language. It is built on top of the ZipperAG (Martins et al., 2013) library, which provides support for Attribute Grammars. The work presented in this paper is inspired by Zstrategic and ZipperAG, and it also showcases that these techniques are valid in languages other than Haskell. The Zstrategic library and our Python pyZstrategic counterpart are inspired by Strafunski (Lämmel and Visser, 2002). There is, however, a key difference between these libraries: while Strafunski accesses the data structure directly, pyZstrategic and Zstrategic operate on zippers. As a consequence, in both zipper-based libraries, the programmer can easily access attributes from strategic functions and strategic functions from attribute equations. Accessing attributes, and thus performing context-dependent transformations/refactorings, are not possible in Strafunski.

The Stratego program transformation language (now part of the Spoofax Language Workbench (Kats and Visser, 2010)) supports the definition of rewrite rules and programmable rewriting strategies, and it is able to construct new rewrite rules at runtime, to propagate contextual information for concerns such as lexical scope. It is argued in (Kramer and Van Wyk, 2020) that contextual information is better specified through the usage of inherited attributes for issues such as scoping, name-binding, and type-checking. We present this same usage of inherited attributes in our attribute grammar examples.

Kiama was developed by Sloane (Sloane et al., 2010; Kats et al., 2009): an embedding of strategic term rewriting and AGs in the Scala programming language. While our approach expresses both attribute computations and strategic term rewriting as pure functions, 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. Such global caching, however, induces an overhead in order to keep it updated, for example, attribute values associated with subtrees discarded by the rewriting process need to be purged from the cache (Sloane et al., 2014). In our setting, which is inspired in a purely functional setting, we only compute attributes in the desired rewritten tree (as is the case of the let example shown in section 3.1).

Influenced by Kiama, Kramer and Van Wyk (Kramer and Van Wyk, 2020) present *strategy attributes*, which is an integration of strategic term rewriting into attribute grammars. Strategic rewriting rules can use the attributes of a tree to reference contextual information during rewriting, much like we present in our work. They present several practical applications, namely the evaluation of λ -calculus, a regular expression matching via Brzozowski derivatives, and the normalization of for-loops. All these examples can be directly expressed in our setting. They also present an application to optimize translation of strategies. Because our techniques

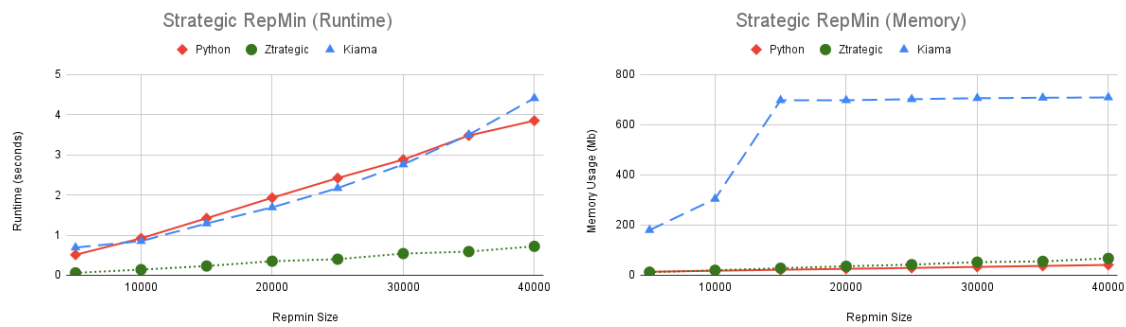


Figure 4: Strategic Repmin: Python versus Ztrategic versus Kiama.

rely on a shallow embedding, we are unable to express strategy optimizations without relying on meta-programming techniques (Sheard and Jones, 2002). Nevertheless, pyZtrategic embedding result in a very simple and concise Python libraries that are easier to extend and maintain, specially when compared with the complexity of extending a full language system such as Silver (Van Wyk et al., 2008).

JastAdd is a reference attribute grammar based system (Ekman and Hedin, 2007). It supports most of AG extensions, including reference and circular AGs (Söderberg and Hedin, 2013). It also supports tree rewriting, with rewrite rules that can reference attributes. JastAdd, however, provides no support for strategic programming, that is to say, there is no mechanism to control how the rewrite rules are applied.

6 CONCLUSIONS

This paper presented pyZtrategic: a Python library that supports the combined embedding of strategic term rewriting and attribute grammars in Python. This is the first library offering the power of strategic term rewriting and attribute grammars to Python programming. This library has been used to support an advanced MSc course on software maintenance and evolution, namely in developing source code analysis and transformation tools.

We showed several examples of using pyZtrategic and comparing its expressiveness and performance with other similar systems, namely Stratego and Ztrategic. Our first results show that pyZtrategic offers the expressiveness of the other libraries, while being its runtime performance affected by the current poor performance of the Python language. PyZtrategic, however, is implemented as a pure embedded DSL in Python and, consequently, is immediately affected by any development in such host language. Thus, we expect a significant increase on the pyZtrategic performance when new Python compilers are widely avail-

able, such as the recently presented high-performance Codon compiler (Shajii et al., 2023).

ACKNOWLEDGEMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020. The second author is also sponsored by FCT grant 2021.08184.BD.

REFERENCES

- Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. (2007). Tom: Piggybacking rewriting on java. In Baader, F., editor, *Term Rewriting and Applications*, pages 36–47. Springer Berlin Heidelberg.
- Cordy, J. R. (2004). Tx1 - a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31.
- Dijkstra, A. and Swierstra, S. D. (2005). Typing Haskell with an attribute grammar. In Vene, V. and Uustalu, T., editors, *Advanced Functional Programming*, pages 1–72. Springer Berlin Heidelberg.
- Ekman, T. and Hedin, G. (2007). The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18.
- Gray, R. W., Levi, S. P., Heuring, V. P., Sloane, A. M., and Waite, W. M. (1992). Eli: A complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130.
- Huet, G. (1997). The Zipper. *Journal of Functional Programming*, 7(5):549–554.
- Kats, L. C. and Visser, E. (2010). The Spoofox Language Workbench: Rules for declarative specification of languages and ides. In *Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 444–463. ACM.
- Kats, L. C. L., Sloane, A. M., and Visser, E. (2009). Decorated attribute grammars: Attribute evaluation meets strategic programming. In *Proceedings of the 18th*

- International Conference on Compiler Construction*, CC '09, page 142–157. Springer-Verlag.
- Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145.
- Kramer, L. and Van Wyk, E. (2020). Strategic tree rewriting in attribute grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 210–229. ACM.
- Kuiper, M. and Saraiva, J. (1998). Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag.
- Lämmel, R. and Visser, J. (2002). Typed combinators for generic traversal. In Krishnamurthi, S. and Ramakrishnan, C. R., editors, *Practical Aspects of Declarative Languages*, pages 137–154. Springer.
- Lämmel, R. and Visser, J. (2003). A strafunski application letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, page 357–375. Springer-Verlag.
- Luttik, S. P. and Visser, E. (1997). Specification of rewriting strategies. In *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications*, Algebraic'97, page 9, Swindon, GBR. BCS Learning & Development Ltd.
- Macedo, J. N., Rodrigues, E., Viera, M., and Saraiva, J. (2024). Zipper-based embedding of strategic attribute grammars. *Journal of Systems and Software*, 211:111975.
- Macedo, J. N., Viera, M., and Saraiva, J. (2022). Zipping strategies and attribute grammars. In *16th International Symposium on Functional and Logic Programming, FLOPS 2022*, page 112–132. Springer-Verlag.
- Martins, P., Fernandes, J. P., and Saraiva, J. (2013). Zipper-based attribute grammars and their extensions. In Du Bois, A. R. and Trinder, P., editors, *Programming Languages*, pages 135–149. Springer Berlin Heidelberg.
- Mernik, M., Korbar, N., and Žumer, V. (1995). Lisa: A tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79.
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609.
- Reps, T. and Teitelbaum, T. (1984). The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48.
- Saraiva, J. (1999). *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, The Netherlands.
- Saraiva, J. (2002). Component-based programming for higher-order attribute grammars. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, pages 268–282.
- Saraiva, J. and Swierstra, D. (1999). Generic Attribute Grammars. In Parigot, D. and Mernik, M., editors, *Second Workshop on Attribute Grammars and their Applications*, WAGA'99, pages 185–204, Amsterdam, The Netherlands. INRIA Rocquencourt.
- Shajii, A., Ramirez, G., Smajlović, H., Ray, J., Berger, B., Amarasinghe, S., and Numanagić, I. (2023). Codon: A compiler for high-performance pythonic applications and dsls. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023*, page 191–202, New York, NY, USA. Association for Computing Machinery.
- Sheard, T. and Jones, S. P. (2002). Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02*, page 1–16, New York, NY, USA. Association for Computing Machinery.
- Sloane, A. M., Kats, L. C. L., and Visser, E. (2010). A pure object-oriented embedding of attribute grammars. *Electronic Notes in Theoretical Computer Science*, 253(7):205–219.
- Sloane, A. M., Roberts, M., and Hamey, L. G. C. (2014). Respect your parents: How attribution and rewriting can get along. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., editors, *Software Language Engineering*, pages 191–210, Cham. Springer International Publishing.
- Söderberg, E. and Hedin, G. (2013). Circular higher-order reference attribute grammars. In Erwig, M., Paige, R. F., and Van Wyk, E., editors, *Software Language Engineering*, pages 302–321, Cham. Springer International Publishing.
- van den Brand, M. G. J., Deursen, A. v., Heering, J., Jong, H. A. d., Jonge, M. d., Kuipers, T., Klint, P., Moonen, L., Olivier, P. A., Scheerder, J., Vinju, J. J., Visser, E., and Visser, J. (2001). 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. Springer-Verlag.
- Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2008). Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116.
- Visser, E. (2001). 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. Springer-Verlag.
- Visser, E., Benaissa, Z.-e.-A., and Tolmach, A. (1998). Building program optimizers with rewriting strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98*, page 13–26, New York, NY, USA. Association for Computing Machinery.