Is There Hypothesis for Attribute Grammars?

Emanuel Rodrigues
HASLab & INESC TEC, University of Minho, Braga, Portugal

José Nuno Macedo
HASLab & INESC TEC, University of Minho, Braga, Portugal

João Saraiva
Department of Informatics, HASLab & INESC TEC, University of Minho, Braga, Portugal

Abstract

Software testing is essential to ensure software reliability and functionality. Numerous testing techniques have been proposed, and most programming languages provide built-in support for testing. However, these techniques have yet to be integrated into the powerful attribute grammar formalism.

This paper introduces a method to incorporate property-based testing into attribute grammars. We achieve this by embedding attribute grammars in *Python* which is combined with its powerful *Hypothesis* property-based testing framework. To validate our approach, we express properties of scope rules in a (nested) block-structured language, commonly found in many (functional) languages.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Theory of computation → Rewrite systems; Software and its engineering → Software testing and debugging

Keywords and phrases Property-based Testing, Attribute Grammars, Strategic Term Rewriting

Digital Object Identifier 10.4230/OASIcs.Programming.2025.19

Supplementary Material *Software (Source Code)*: `https://tinyurl.com/sclit25tools` [16]

Funding 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.

1 Introduction

Software testing is a vital activity in the development life cycle of software systems. In fact, most programming languages and paradigms offer powerful testing frameworks to evaluate and improve the quality of software systems. In the context of software language engineering, Knuth introduced attribute grammars as a powerful programming paradigm for specifying the static semantics of languages. However, their expressiveness has extended their use to various other domains, such as to express complex multiple traversal tree-based algorithms, to specify advanced type inference algorithms, to elegantly implement pretty printing functions, to strictify lazy circular functions, etc.

Surprisingly, no attribute grammar-based system provides a testing framework. In this paper, we incorporate property-based testing, expressed via the Hypothesis system, in the pyZtrategic attribute grammar-based system [15]: a combined embedding of attribute grammars and strategic term rewriting defined in *Python*. As a result, we provide an advanced mechanism for attribute grammar writers to test their specifications by defining properties on attributes. Strategic term rewriting is a key component of our approach: attribute properties are tested on all attribute instances of an abstract syntax tree through strategic functions. Moreover, we use the generator mechanism provided by Hypothesis to

automate the generation of test cases, *i.e.*, abstract syntax trees. To validate the proposed technique, we consider an attribute grammar expressing advanced scope rules of a block structured language, and we show several properties defined over its attributes.

2 Background

This section includes a brief introduction to attribute grammars, property-based testing, and strategic term rewriting. We will present these techniques in the *Python* language, in which we specify a powerful combined embedding of these three domain-specific techniques.

Before we present each of these powerful programming techniques, we start by presenting a simple but motivating example that we will use throughout the paper. Consider the well-known *Repmin* program. The goal of this program is to transform a binary leaf tree[1] of integers into a new tree with the exact same shape but in which all leaves have been replaced by the minimum leaf value of the original tree.

To model such a binary tree, we consider the following *Python* algebraic data type:

```
type Tree = Fork            @dataclass                    @dataclass
          | Leaf            class Fork:                   class Leaf:
                                left: Tree                    leaf: int
                                right: Tree
```

`Leaf` and `Fork` are data type constructors, which are functions used to construct instances of binary leaf trees of type `Tree`. For example, we can define a tree, t_1, in *Python* as follows:

```
t1=Fork(Leaf(3), Fork(Leaf(2), Leaf(4)))
```

Next, we define the recursive function `tmin` to compute a tree's minimum value. Note that, while *Python* is not a typed language, it allows type annotations which we add to make functions more readable.

```
def tmin(t: Tree) -> int:
    match(t):
        case Leaf(n)  :  return n
        case Fork(l,r):  return min(tmin(l), tmin(r))
```

Where `min` computes the minimum of two given numbers. Similarly, we can express the recursive function `replace`, which places a given concrete value in all leaves of a tree:

```
def replace(t: Tree, n: int) -> Tree:
    match(t):
        case Leaf(_)  :  return Leaf(n)
        case Fork(l,r):  return Fork(replace(l,n), replace(r,n))
```

Now, we combine functions `replace` and `tmin` to obtain a simple solution to *Repmin*:

```
def repmin(t: Tree) -> Tree:
    return replace(t, tmin(t))
```

Regarding this implementation, the underlying tree `t` is traversed twice: once to compute the minimum value and a second time to replace the value stored in the leaves by the computed minimum. In fact, there are two recursive functions with `t` as argument.

[1] A tree that stores its values in the leaves.

2.1 Attribute Grammars

Let us now express *Repmin* within the Attribute Grammar (AG) formalism. Instead of presenting the formal AG definition of *Repmin*[2], we will adopt a visual notation that is often used by AG developers to sketch a first draft of their grammars. We combine the visual representation of the AG with its definition in the well-known Synthesizer Specification Language (SSL) [14]. SSL is a Domain Specific Language (DSL) for expressing AGs that is used by the Synthesizer Generator [13] and by the *LRC* [10] AG-based systems.

2.1.1 Types as Grammars

Grammars specify formal languages, where a formal language consists of a (possibly non-finite) set of sentences. Similarly, types define a (possibly non-finite) set of values. A parser of the language (usually generated from its grammar) checks whether a given input is a sentence of the language, *i.e.* if it is in its set of sentences. In functional programming, a type inference mechanism produces a type checker that checks if a value is type-correct, *i.e.* if it is in the set defined by the (inferred) type. Although context-free grammars are widely used to specify syntactic parsers, in the AG formalism they are also used to define abstract tree structures. Thus, grammars can be seen as types, where a type corresponds to a non-terminal symbol, and a type constructor to a production. Thus, in *Repmin* `Tree` is a non-terminal and `Leaf` and `Fork` are productions (names).

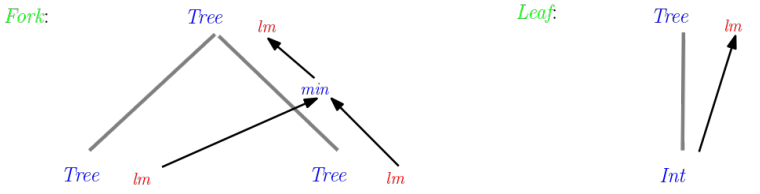
In SSL we need to define the (abstract) grammar, very much like the *Python* abstract data types we introduced for *Repmin*.

```
1 Tree : Leaf (INT)
2      | Fork (Tree Tree);
```

2.1.2 Function Results as Synthesized Attributes

Note that in *Repmin* we need to compute the minimum number stored in the leaves of the tree. In AGs, we associate instances of attributes with tree nodes that are used to synthesize such values. *Synthesized attributes* propagate the attribute values upward in the tree.

Thus, to express the computation of the minimum value we define a synthesized attribute, with name `lm` (local minimum) of type `Int`, that we associate to non-terminal `Tree`. Next, we present the Visual AG fragment that represents the equations that define this synthesized attribute for the two productions of `Tree`.



In this notation, occurrences of synthesized attributes are shown on the right of their non-terminals, and we omit their types. Moreover, the attribute equations of the productions are defined with upward arrows as expressed by the synthesized attributes. For example, in constructor/production `Fork`, the `min` function gets as arguments the synthesized minimum of the two subtrees of type `Tree` (corresponding to the two right-hand side non-terminals

[2] The interested reader can find such specification in [17].

of the production), in order to synthesize the minimum of the tree (*i.e.*, the left-hand side non-terminal, or the "parent"). For the `Leaf` production, the synthesized minimum is defined as the `Int` value stored in the leaf.

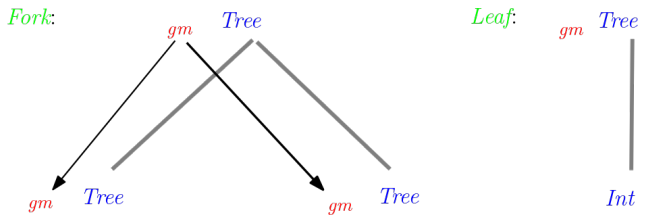
In the SSL notation, the definition of the rules specifying the synthesized attribute `lm` is written as follows:

```
1 Tree { syn INT lm; };
2 Tree : Leaf { Tree.lm   = INT;                        }
3      | Fork { Tree$1.lm = min(Tree$2.lm,Tree$3.lm); };
```

As expected, the SSL notation directly follows the visual notation presented before.

2.1.3 Function Arguments as Inherited Attributes

Having synthesized the minimum number of the *Repmin* tree, we need to pass it downwards to the leaves, in order to compute the new tree with that number stored in the leaves. AGs use *inherited attributes* to pass information downwards the tree. Thus, we introduce an inherited attribute, named `gm` (global minimum) of type `Int`, which is inherited by all `Tree` nodes (symbols). We display occurrences of inherited attributes on the left of their non-terminals. In this case they are just copied downwards the tree, as shown next:



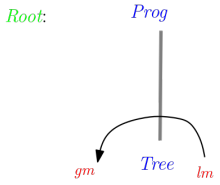
In the SSL notation, the specification of rules defining the inherited attribute `gm` are defined as follows:

```
1 Tree {  inh INT gm; };
2 Tree : Fork  { Tree$2.gm = Tree$1.gm;
3                Tree$3.gm = Tree$1.gm; };
```

In order to express within the AG formalism that the global minimum `gm` of the *Repmin* tree is the local minimum `lm` of the root of the tree, we introduce a new "root" non-terminal symbol/type `Prog`:[3]

```
1 root Prog;              /* root of the grammar */
2 Prog : Root (Tree);
```

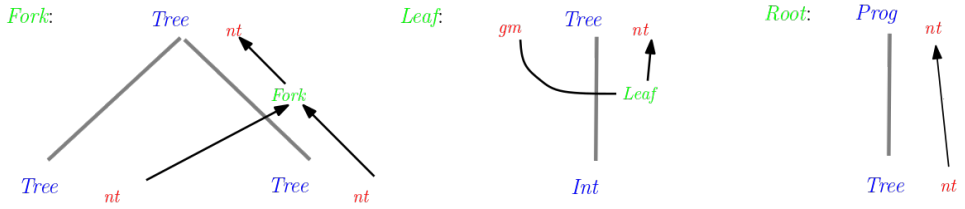
In its production/constructor `Root`, the `Tree` corresponds to the root of the *Repmin* tree. `Tree` has two attributes already: the inherited `gm` and the synthesized `lm`, which we display on its left/right side, respectively. Now, we can express the equation stating that the global minimum of that non-terminal symbol is the synthesized attribute of that same symbol.




```
1 Prog : Root { Tree.gm = Tree.m; };
```

[3] The root non-terminal symbol of an AG does not have inherited attributes [8]. In other words, (attribute) grammars are context-free.

We can now synthesize the desired new tree (`nt`). In a `Fork` production, the new tree of the left subtree and the new tree of the right subtree are combined in a new `Fork` tree. For a `Leaf`, a new `Leaf` is constructed that contains the inherited global minimum `gm`.



The equations for synthesized attribute `nt` in SSL look as follows:

```
1 Tree , Prog { syn Tree nt;  };
2 Tree : Leaf { Tree.nt   = Leaf(Tree.gm);             }
3      | Fork { Tree$1.nt = Fork(Tree$2.nt,Tree$3.nt); };
4 Prog : Root { Prog.nt = Tree.nt;                     };
```

We have just presented the complete AG that specifies the *Repmin* problem. This style of programming with attribute grammars has three key features:

- Firstly, from such a specification, AG-based systems generate programs - the so-called attribute evaluators - that traverse the underlying abstract syntax tree of a given *Repmin* program while computing instances of the attributes.
- Secondly, in the elegant style of AG programming, programmers do not have to specify (recursive) traversal functions - like the *Python* functions `tmin` and `replace` presented before - nor to schedule such traversals - as defined in the *Python* `repmin` function. In fact, the *Repmin* problem is a simple example, where the implementation and scheduling of recursive (traversal) functions is straightforward. However, there are language engineering algorithms that rely on a large number of traversals whose traversal functions are intertwined. Thus, scheduling traversals is a complex task. Within the AG formalism, AG systems based on dependencies among attributes (induced by the AG equations) use powerful scheduling algorithms [5] that both compute an evaluation order and detect (real) circular dependencies. If no circular dependencies arise, then an efficient and always terminating multiple-traversal attribute evaluator is generated [6, 17]. In fact, the *Python* solution of the *Repmin* problem can be seen as the (*Python*) attribute evaluator generated by an AG system of the *Repmin* AG[4].
- Finally, many language engineering algorithms, such as name analysis [19], type inference/checking [3, 12], advanced pretty printing [18], rely on multiple traversals that compute information in visits to nodes of the AST that are needed in the following visits. This information needs to be explicitly passed between traversals, either by storing the information in the tree's node (as a side effect of the traversal function, as typical in imperative programming) or by using additional gluing data structures [17] (as usual in declarative programming). In fact, such algorithms are hard and complex to write in general-purpose languages. In contrast, they are elegantly and straightforwardly specified in the style of AG programming.

[4] The *LRC* system, given the presented SSL specification of *Repmin* as input, generates *Haskell* and *Ocaml* evaluators equivalent to our *Python* solution. *LRC* is a retargetable system, and such a *Python* back-end can be easily implemented in the system.

In Section 4 we will present a simple algorithm for name analysis that relies on complex and intertwined traversals that in a straightforward implementation would rely on additional data structures to convey information between traversals.

2.2 Property-based Testing

Writing tests is essential to ensure the high quality of the software under test. Software testing is usually performed by writing *unit tests*: small test cases that each test a small piece of functionality of your code. While it is true that writing unit tests is important, writing unit tests manually is time-consuming and complex. It is time-consuming because it involves the need to write many separate unit tests for each piece of functionality, which all look more or less the same. And it is complex because it is very easy to miss a certain combination of inputs for which the program does not behave as expected.

Property-based Testing (*PbT*) follows a different approach: instead of writing unit tests, the programmer writes down properties expected to hold for his program. These properties are validated by a *PbT* framework by automatically generating many different random input values, and then checking that the property holds for all these combinations of values. *PbT* was proposed by Claessen and Hughes in their seminal work on the *Haskell* library *QuickCheck* [1]. Soon after, the power of property-based testing was incorporated in most programming languages. *Python* is no exception: *Hypothesis* offers a powerful system for property-based testing in *Python*. Next, we briefly present some properties and generators for *Repmin* to show the power of this testing technique.

Before we write some properties for the *Repmin* program, let us define two functions on binary leaf trees: the function `count` counts the number of leaves/values of a given tree,

```
def count(t: Tree) -> int:
    match(t):
        case Leaf(l)  :  return 1
        case Fork(l,r):  return count(l) + count(r)
```

while the function `toList` produces a list containing all values of the given tree.

```
def toList(t: Tree) -> list[int]:
    match(t):
        case Leaf(l)  :  return [l]
        case Fork(l,r):  return toList(l) + toList(r)
```

Next, we define a property as a regular Python boolean function that checks whether the number of leaves in a tree is equal to the number of leaves in the tree produced by `repmin`.

```
def prop_repmin1(t: Tree) -> bool:
    return count(t) == count(repmin(t))
```

We define a property stating that the number of values in tree `t`, multiplied by its minimum value, equals the sum of all values in the tree produced by *Repmin* for `t`.

```
def prop_repmin2(t: Tree) -> bool:
    return (count(t) * tmin(t)) == sum(toList(repmin(t)))
```

2.2.1 Random Input Generators

A key ingredient of property-based testing is the automatic generation of (many) random inputs to validate properties. Next, we define a generator in *Hypothesis* for the `Tree` data type. Here, `integers` is a predefined random integer generator, `one_of` selects one generator randomly from a list of generators, and `draw` is a function internally used to extract values from generators adequately.

```python
@composite
def genLeaf(draw) -> Leaf:
    i = draw(integers(min_value=1))
    return Leaf(i)
@composite
def genFork(draw) -> Fork:
    return Fork(draw(genTree()), draw(genTree()))
@composite
def genTree(draw) -> Tree:
    return draw(one_of([genLeaf(), genFork()]))
```

We associate the *Hypothesis* generator with the assertion of the two properties as follows:

```python
@given(genTree())
def test_prop_repmin1(i):
    assert prop_repmin1(i)
@given(genTree())
def test_prop_repmin2(i):
    assert prop_repmin2(i)
```

The property-based testing *Hypothesis* system offers an advanced testing framework to test our *Python* functions/programs. Unfortunately, it is not possible to integrate *PbT* to test AG specifications such as the *Repmin* AG shown in Section 2.1.

To combine AGs and *PbT* we consider the technique we proposed in [2] in the context of the *Haskell* programming language. Thus, we rely on strategic term rewriting [11] to apply a property expressed on attributes to all nodes in the AST. Next, we briefly present this powerful AST transformation technique.

2.3 Strategic Term Rewriting

The *Stratego* system [20], developed by Eelco Visser, played a key role in the development of strategic term rewriting. *Stratego* is a powerful and widely used strategic term rewriting system that is part of the Spoofax Language Designer's Workbench [7]. *Stratego* uses a domain-specific language to express algebraic data types, rewriting rules, and strategies to apply these rules. It follows a traditional architecture, including a language processor that compiles specifications into efficient strategic programs (expressed in C). However, such systems tend to be large, complex, and challenging to maintain and evolve.

In this section, we show *pyZtrategic* which provides an embedding of strategic term rewriting that does not rely on such a large language system [15]. Instead, it uses techniques to embed a DSL - defining a strategic program - directly into *Python*. *pyZtrategic* offers *Python* programmers recursion schemes (strategies) which apply term rewrite rules in defining large scale language transformations. Next, we define a *full top-down* strategy that increments the integers stored in the leaves of the *Repmin* tree.

```python
def inc(t: Tree) -> Tree:
    def aux(tr: Tree) -> Tree:
        match(tr):
            case Leaf(l):  return Leaf(l+1)
            case _       :  raise st.StrategicError
    return st.full_tdTP(lambda x: st.adhocTP(st.idTP, aux, x),obj(t)).node()
```

The previous strategic-based function expresses a type-preserving transformation. Thus, the type of the input tree is preserved. In type-preserving transformations (identified by the suffix `TP` in the strategy's name) the resulting tree has exactly the same type as the input

one. *pyZtrategic* provides similar term rewriting strategies that do not preserve the type of the input AST. These strategies are called type-unifying transformations (identified by the suffix `TU` in the strategy's name). Next, we use a *full bottom-up* strategy to produce the list of all integers in the tree (then, the `sum` function computes its total).

```
def sumTree(t: Tree) -> int:
    def aux(tr: Tree) -> list[int]:
        match(tr):
            case Leaf(x):  return [x]
            case _       :  return []
    return sum(st.full_buTU(lambda x: st.adhocTU(st.failTU, aux, x), obj(t)))
```

3 Property-based testing for Attribute Grammars

Combining property-based testing systems with attribute grammar systems is not straightforward. This challenge arises because properties are typically expressed in general-purpose programming languages, rather than in the domain-specific languages used by AG systems. One possible solution would be to extend the AG DSL to support *PbT*. However, this would require developing a new *PbT* framework specifically designed for AGs, which would also need to be integrated into the attribute evaluators' functions. It should be noticed that in Section 2.2 we expressed properties on the *Repmin* program that can be seen as the attribute evaluator that an AG system would generate from the *Repmin* AG shown in Section 2.1. However, the *Python* code was manually written, rather than being produced by an AG system. Of course, programmers should write properties on the code/specification they write and not on generated code. In fact, the evaluators generated from AG systems, defining real language engineering problems, are complex and difficult to understand.

To integrate attribute grammars and property-based testing, we propose a novel methodology that incorporates the following techniques:

- First, we propose a novel embedding of attribute grammars and strategic term rewriting in the *Python* programming language, as provided by the *pyZtrategic* library [15]. *pyZtrategic* offers an shallow Embedded Domain-Specific Language (EDSL) that enables AGs and strategic term rewriting to be specified directly as regular Python programs. In this setting, the Embedded Attribute Grammar (EAG) is an executable *Python* program: given an AST it computes the values of its attribute instances.
- Second, properties are expressed directly on the attributes of the AG using the *Hypothesis* library. Since AGs are embedded in *Python*, properties have direct access to these attributes. As a result, writing properties for AG attributes closely resembles defining properties for regular *Python* programs.
- Third, property-based testing systems include mechanisms for the automatic generation of (many) random inputs to validate properties. *Hypothesis* is no exception. Thus, we will rely on such *Hypothesis* generators to produce ASTs and then check that properties on attributes instances hold for all the ASTs decorated by the *Python* EAG.
- Finally, we rely on strategic programming to apply these properties to all instances of attributes in a given decorated abstract syntax tree[5].

[5] A decorated AST is a tree in which the values of all attribute instances have been computed.

3.1 *pyZtrategic*: Embedding AGs and Strategies in *Python*

Before we define properties on attributes of an AG, let us discuss our *pyZtrategic* system that provides an elegant embedding of AGs in *Python* [15]. Our embedding relies on an advanced data structure to navigate on heterogenous ASTs: the Zipper[6] data structure [4]. In fact, it is the Zipper mechanism that supports the (dynamic) evaluation of attributes by navigating upwards/downwards in the AST while computing (on demand) attributes.

To show the expressiveness of *pyZtrategic*, we return to our simple running example. In the *Repmin* AG we considered a root symbol that does not have inherited attributes as required by the AG formalism[7]. Thus, we extend the data type (presented in Section 2) that defines the abstract grammar of *Repmin*:

```
type Tree = Root            @dataclass
          | Fork            class Root:
          | Leaf                tree: Tree
```

Having defined the (abstract) grammar of the *Repmin* problem, now we need to define the *Repmin* attributes and their equations. We start by writing directly in *Python* the AG equations that define the synthesized attribute `lm`. In *pyZtrategic* we express it in a zipper-based function with attribute's name, `lm` in this case. We use a `constructor` function to peek into the zipper and return a representation of the production the attribute is applied to - for a `Leaf`, the `lexeme` syntactic reference will return the value stored in the leaf, and for a `Fork`, we compute the minimum of the `lm` of the children which we obtain using the accessor function `z_dollar`.

```
def lm(z: Zipper[Tree]) -> int:
    match constructor(z):
       case Constructor.CLeaf: return lexeme(z)
       case Constructor.CFork: return min(lm(z.z_dollar(1)), lm(z.z_dollar(2)))
       case _                : raise st.StrategicError
```

This *Python* function follows the visual and SSL definitions of the attribute equations presented in Section 2.1.2. Next, we define the attribute `gm` as the following *Python* function:

```
def gm(z: Zipper[Tree]) -> int:
    match constructor(z):
        case Constructor.CRoot:  return lm(z.z_dollar(1))
        case Constructor.CLeaf:  return gm(z.up())
        case Constructor.CFork:  return gm(z.up())
```

Once again, the *Python* function `gm` directly follows the AG definitions in Section 2.1.3. Please note that in this *Repmin* EAG we are defining that in `Root` production the global minimum `gm` is the local minimum `lm` of the first child (as shown in Section 2.1.3). The construction of the new tree is expressed by the following synthesized attribute/function `nt`:

```
def nt(z: Zipper[Tree]) -> Tree:
    match constructor(z):
       case Constructor.CRoot: return Root(nt(z.z_dollar(1)))
       case Constructor.CLeaf: return Leaf(globmin(z))
       case Constructor.CFork: return Fork(nt(z.z_dollar(1)), nt(z.z_dollar(2)))
```

[6] Zippers are also implemented as a Python library `https://pypi.org/project/zipper/`.
[7] By definition of an AG the root symbol is context free.

Finally, we define *Repmin* as the `nt` of a tree after moving it into a zipper with the zipper function `obj`:

```
def repminAG(t: Tree) -> Tree:
    return nt(obj(t))
```

We extend our *Repmin* generator to include the `Root`:

```
@composite
def genTreeRoot(draw) -> Root:
   return Root(draw(genTree()))
```

We define properties expressed on all attribute instances of a specific production:

```
@given(genTreeRoot())
def testPropLocMin(i):
    assert all(st.full_tdTU(lambda x: st.adhocTUZ(st.failTU,validateLocMin, x), obj(i)))
```

Testing that the integer stored in the leaves is always greater or equal to the global minimum values:

```
def validateLocMin(t: Tree, z: Zipper[Tree]) -> list[bool]:
    match(t):
        case Leaf(l):  return [l >= gm(z)]
        case _       :  return []
```

We can define a property that performs this test in all productions of the grammar:

```
def validateLocMin(t: Tree, z: Zipper[Tree]) -> list[bool]:
    match(t):
        case Leaf(l):  return [l >= gm(z)]
        case Fork() :  return [lm(z) >= gm(z)]
        case Root() :  return []
```

We can also define properties on the synthesized attributes of the Root symbol: the results of the attribute evaluator. Next, we define a property to verify whether the global minimum of the original tree is equal to the global minimum of the tree.

```
@given(genTreeRoot())
def testGlobminPreserved(t):
    assert gm(obj(t)) == gm(obj(repminAG(t)))
```

4 Testing Properties of Scope Rules

Names serve to identify program entities such as variables, functions, types, and modules, enabling these entities to be referenced throughout a program. Name resolution determines the association between each reference and its corresponding declaration(s) based on the language's scope rules. Specification of such scope rules is often complicated because it cuts across the local inductive structure of programs (as described by an abstract syntax tree). For example, the name introduced by a *let* node in an AST representing a functional program may be referenced by a node arbitrarily distant from its declaration. Attribute grammars offer a powerful way to define complex scope rules in a structured and declarative setting. In fact, name analysis played a foundational role in Donald Knuth's pioneering work on AGs [9].

To express the scope rules of programming languages, we will consider the (sub)language of let expressions as incorporated in most functional languages, such as *Haskell*. Although being a concise example, the *Let* language holds the central characteristics of programming

languages, such as (nested) block-based structures and mandatory but unique name declarations. In addition, the semantics of *Let* does not force a declare-before-use discipline, meaning that a variable can be declared after its first use. The following is an example of a program in the *Let* language, which corresponds to the correct *Haskell* code.

```
program =  let  a = b + 3
               c = 2
               b = c * 3 - c
           in   (a + 7) * c
```

This expression evaluates to `28`. Usually, *Let* expressions allow nesting which complicates the scope rules: nested let expressions inherit the context (*i.e.*, declared names) of their outermost ones, where nested declarations hide outer ones. Next, we show an example where the nested declaration of the name `a` hides its outer declaration.

```
ex = let a = 2 + c          -- 1: dcli = []
         b = let a = 3 + c  -- 3: dcli = [(a,0),(b,0),(c,0)]
             in  a          -- 4: dclo = [(a,0),(b,0),(c,0),(a,1)]
         c = 2 + w
         a = 3
     in a * 5               -- 2: dclo = [(a,0),(b,0),(c,0)]
```

The *Let* language does not force a *declare-before-use* discipline: This is the case in the first use of `c`: it refers to its (later) definitions in the outermost block. Note that nested *Let* defines a second name `a`. Consequently, the use of name `a` (in the `in` part of the nested let) refers to the inner definition and not to the outer definition. However, in a `let` block, a name may be declared once, at most. So, the second definition of the identifier `a` in the outermost block is invalid. Furthermore, the *Let* language requires that only defined names may be used. As a result, the applied occurrence of `w` is invalid, since `w` has no binding occurrence at all. Unused names, such as `b`, are not errors, although, they represent dead code.

We aim to develop a program that analyzes *Let* programs and generates a list of identifiers that violate the language's scope rules. To simplify identifying incorrectly used names in a *Let* program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing this sentence is `[b,w,a]`.

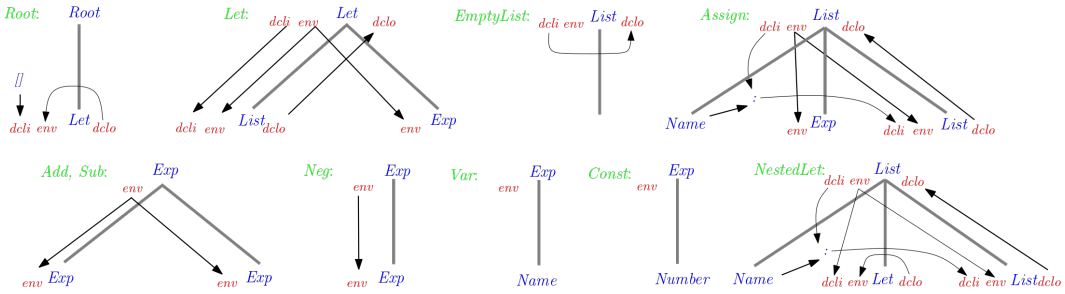
A conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for accumulating the declarations of names and constructing an environment and a second time to process the uses of names (using the computed environment) in order to check for the use of undeclared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier

1^{st} *Traversal*	2^{nd} *Traversal*
▬ *Collect the list of local definitions*	▬ *Use the list of definitions as the global environment*
▬ *Detect duplicate definitions (using the collected definitions)*	▬ *Detect use of non defined names*
	▬ *Combine "both" errors*

As annotated in the *Let* program `ex` (using Haskell comments), the outermost `let` block is context-free. It begins by accumulating the declared names in an empty list of declarations (`dcli` in the comment labelled `1`). This accumulated list forms the environment of the outer let (`dclo` in comment `2`). For a nested *Let*, the initial context – its starting list of declarations – is the total environment of the enclosing *Let* (see `dcli` in comment `3`). To distinguish declarations with the same name at different nesting levels, we pair each name with its corresponding level.

In such a straightforward algorithm, semantic errors caused by duplicate definitions are detected during the first traversal, whereas errors due to missing declarations are identified in the second traversal. Consequently, errors found in the first traversal must be explicitly passed to the second.[8] Additionally, scheduling the traversal functions is complex: the first traversal of inner blocks can only begin during the second traversal of the outer block. As a result, the traversal functions become intertwined. In the elegant style of AG programming, the grammar writer neither needs to schedule complex traversal functions nor rely on additional mechanisms to pass information between traversals. The AG system automatically infers these traversals and produces the necessary mechanisms to convey information between traversal functions.

Next, we show the visual representation of the AG that specifies the scope rules of the *Let* language.[9]



In these attribute equations, we can see that attributes `dcli` (inherited) and `dclo` (synthesized) are used to define an accumulator: it starts at the root as the empty list, and at `Assign` and `Nested` productions, the Name is added to that list. The inherited attribute `env` is used to pass downwards the accumulated list as the environment. It should be noticed that at `NestedLet` productions, such environment (`env`) is copied to the initial list of declarations of the nested let (attribute `dcli`). In the nested *Let* the environment `env` is the accumulated list of declarations of the same *Let* (attribute `dclo`). Having expressed this non-trivial scope rules in the elegant AG formalism, we now write several properties to test our specification.

1. Property stating that every declared name needs to be in the environment:

```
def checkOneDeclared(t: Lets, z: Zipper[Lets]) -> list[bool]:
    match t:
        case Assign(x, _, _) | NestedLet(x, _, _):  return [mBIn(x, env(z))]
        case _                                    :  return []
def testDeclaredNames(i):
    assert all(st.full_tdTU(lambda x: st.adhocTUZ(st.failTU, checkOneDeclared, x),
    ↪  obj(i)))
```

   Where `mBIn` is a simple lookup function that returns true if the name (its first argument) is in the environment `env` (its second argument). The property is applied to all `Assign` and `NestedLet` nodes/productions using a *full top-down* strategy. The property holds if it is true in `all` nodes.

2. Property stating that every declared name must not be in the accumulated environment thus far. This would correspond to a duplicated declaration.

[8] This is typically achieved by using intrusive code into the implementation of scope rules – either by storing values directly in the AST (as a side effect in imperative programming) or by using gluing data structures to pass such values between traversal functions (as required in a purely functional setting) [17]

[9] Due to space limitations we omit here its definition as a *Python* EAG.

```
def checkOneDeclaredDcli(t: Lets, z: Zipper[Lets]) -> list[bool]:
    match t:
        case Assign(x, _, _) | NestedLet(x, _, _): return [mNBIn((x, lev(z)),
        ↪ dcli(z))]
        case _                                   : return []
def testDeclaredNamesDcli(i):
    assert all(st.full_tdTU(lambda x: st.adhocTUZ(st.failTU, checkOneDeclaredDcli, x),
    ↪ obj(i)))
```

Where `mNBIn` checks whether a given name already exists at the same nested level (given by attribute `lev`) in the accumulated list of declarations `dcli`.

3. Property stating that the initial list of declarations of a nested *Let* is a subset of the total environment of its outer one:

```
def checkNested(t: Lets, z: Zipper[Lets]) -> list[bool]:
    match t:
        case NestedLet(_, _, _):  return [isSubset(dcli(z.z_dollar(2)),
        ↪ env(z.z_dollar(2)))]
        case _                 :  return []
def testNested(i):
    assert all(st.full_tdTU(lambda x: st.adhocTUZ(st.failTU, checkNested, x),
    ↪ obj(i)))
```

In all `NestedLet` nodes, the initial list of declarations (attribute `dcli`) of a nested *Let* (the second child of production/constructor `NestedLet`) is a subset of the total list of declarations (attribute `dclo`) of that same child.

4. Property that states that every declared name is not dead:

```
def checkOneDead(t: Lets, z: Zipper[Lets]) -> list[bool]:
    return [dead(z)]
def testDeadCode(i):
    assert allFalse(st.full_tdTU(lambda x: st.adhocTUZ(st.failTU, checkOneDead, x),
    ↪ obj(i)))
```

Where `dead` is an attribute that checks whether a name declaration is unused.

5. A property that checks whether an optimization preserves the semantics of *Let* programs. In [15] we have expressed the optimization of arithmetic expressions (simplifying multiplication by zero/one, etc) as the innermost strategic function `opt`. The evaluation of a *Let* program is synthesized by attribute `eval`. Thus we write a property that checks if the original AST is equal to the value of the transformed AST.

```
def testEvalInOpt(ast):
    assert eval(obj(ast)) == eval(opt(obj(ast)))
```

This property shows the expressiveness of our approach: It is defined on attributes (`eval`) of the original AST and the AST produced by a strategic-based function (`opt`).

5 Conclusions

This paper presents a technique for expressing property-based testing within the attribute grammar formalism. We combined a *Python* embedding of attribute grammars with the property-based testing framework *Hypothesis*. In this embedding, AGs are written directly as Python functions, allowing *Hypothesis* properties to be expressed directly on attributes. Strategic term rewriting is a key component of our approach: attribute properties are tested on all attribute instances of an abstract syntax tree through strategic functions. To

demonstrate the expressiveness of our technique, we presented several properties to test the scope rules of a modern block-structured *Let* language, as commonly found in many (functional) languages.

Replication Package. All source code discussed in this work, as well as supporting code, is available at `https://tinyurl.com/sclit25tools`.

References

1 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.

2 José Nuno de Macedo, Marcos Viera, and João Saraiva. Property-based testing of attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Int. Conf. on Software Language Engineering*, SLE 2025. ACM, June 2025.

3 Atze Dijkstra and D. Swierstra. Typing Haskell with an attribute grammar. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, pages 1–72. Springer, 2005.

4 Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. doi:10.1017/S0956796897002864.

5 Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980. doi:10.1007/BF00288644.

6 Uwe Kastens. Implementation of Visit-Oriented Attribute Evaluators. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 114–139. Springer-Verlag, 1991. doi:10.1007/3-540-54572-7_4.

7 Lennart C.L. Kats and Eelco Visser. The Spoofax 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, pages 444–463. ACM, 2010. doi:10.1145/1869459.1869497.

8 Donald E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. doi:10.1007/BF01692511.

9 Donald E. Knuth. The genesis of attribute grammars. In *Attribute Grammars and their Applications*, pages 1–12, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. doi:10.1007/3-540-53101-7_1.

10 Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In *7th Int. Conf. on Compiler Construction*, LNCS, pages 298–301. Springer, 1998. doi:10.1007/BFB0026440.

11 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.

12 Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Iterative type inference with attribute grammars. In *Proceedings of the Ninth Int. Conf. on Generative Programming and Component Engineering*, GPCE '10, pages 43–52, New York, NY, USA, 2010. ACM. doi:10.1145/1868294.1868302.

13 T. Reps and T. Teitelbaum. The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48, April 1984. doi:10.1145/800020.808247.

14 T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer, 3 edition, 1989. doi:10.1007/978-1-4613-9633-8.

15 Emanuel Rodrigues, José N. Macedo, Marcos Viera, and João Saraiva. pyZtrategic: A zipper-based embedding of strategies and attribute grammars in python. In *Proc. of the 19th Int. Conf. on Evaluation of Novel Approaches to Software Engineering*, pages 615–624, 2024. doi:10.5220/0012704000003687.

16 Emanuel Rodrigues, José Nuno Macedo, and João Saraiva. pyZtrategic. Software (visited on 2025-09-03). URL: `https://tinyurl.com/sclit25tools`, `doi:10.4230/artifacts.24659`.

17 João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, The Netherlands, December 1999.

18 S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206. Springer, 1999. `doi:10.1007/10704973_4`.

19 S. Doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, January 2009. `doi:10.1017/S0956796808006990`.

20 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proc. of the 12th Int. Conf. on Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–362. Springer-Verlag, 2001. `doi:10.1007/3-540-45127-7_27`.