Delft University of Technology
Software Engineering Research Group
Technical Report Series

Decorated Attribute Grammars. Attribute Evaluation Meets Strategic Programming (Extended Technical Report)

Lennart C. L. Kats, Anthony M. Sloane, Eelco Visser

Report TUD-SERG-2008-038a

TUD-SERG-2008-038a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

This paper is a pre-print of:

Lennart C. L. Kats and Anthony M. Sloane and Eelco Visser. Decorated Attribute Grammars. Attribute Evaluation meets Strategic Programming. In Proceedings of the 18th International Conference on Compiler Construction (CC 2009). Lecture Notes in Computer Science. York, United Kingdom, March 2009.

```
@InProceedings{KSV09,
  author = {Lennart C. L. Kats and Anthony M. Sloane and Eelco Visser},
  title =  {Decorated Attribute Grammars. {A}ttribute Evaluation
            Meets Strategic Programming},
  booktitle = {Proceedings of the 18th International Conference on
               Compiler Construction (CC 2009)},
  month = {March},
  year = {2009},
  series = {Lecture Notes in Computer Science},
  address = {York, United Kingdom},
  publisher = {Springer-Verlag}
  OPTeditor = {},
  OPTpages =  {},
  OPTvolume = {},
}
```

Decorated Attribute Grammars

Attribute Evaluation Meets Strategic Programming

Lennart C. L. Kats[1], Anthony M. Sloane[2,1], Eelco Visser[1]

[1]Software Engineering Research Group, Delft University of Technology,
The Netherlands, L.C.L.Kats@tudelft.nl, visser@acm.org
[2]Department of Computing, Macquarie University,
Sydney, Australia, Anthony.Sloane@mq.edu.au

Abstract. Attribute grammars are a powerful specification formalism for tree-based computation, particularly for software language processing. Various extensions have been proposed to abstract over common patterns in attribute grammar specifications. These include various forms of copy rules to support non-local dependencies, collection attributes, and expressing dependencies that are evaluated to a fixed point. Rather than implementing extensions natively in an attribute evaluator, we propose *attribute decorators* that describe an abstract evaluation mechanism for attributes, making it possible to provide such extensions as part of a library of decorators. Inspired by strategic programming, decorators are specified using generic traversal operators. To demonstrate their effectiveness, we describe how to employ decorators in name, type, and flow analysis.

1 Introduction

Attribute grammars are a powerful formal specification notation for tree-based computation, particularly for software language processing [27], allowing for modular specifications of language extensions and analyses. At their most basic, they specify declarative *equations* indicating the functional relationships between *attributes* (or properties) of a tree node and other attributes of that node or adjacent parent and child nodes [20]. An *attribute evaluator* is responsible for scheduling a tree traversal to determine the values of attributes in a particular tree.

Attribute grammars are nowadays employed in a wide range of application domains and contexts. To extend their expressiveness for use in particular domains, and to abstract over commonly occurring patterns, basic attribute grammars have been extended in many ways, in particular supporting attribution patterns with non-local dependencies. For example, *remote attribution* constructs allow equations that refer to attributes of nodes arbitrarily far above or below the node for which they are defined [6,17]. *Chain attributes* express a dependence that is threaded in a left-to-right, depth-first fashion through a sub-tree that contains definitions of the chain value [17]. *Self rules* provide a local copy of subtrees, which may be adapted for tree transformations [3]. More generally, *collection attributes* enable the value of an attribute of one node to be determined at arbitrary other nodes [6,23]. A different kind of remote attribute is provided by *reference attribute grammars* that allow references directly to arbitrary non-local nodes and their attributes [14], allowing for attributes that look up a particular node or collec-

tion of nodes. Finally, some attribute grammar systems support equations with *circular dependencies* that are evaluated to a fixed point [5,24].

All of these extensions aim to raise the level of abstraction in specifications, by translation into basic attribute grammars or by using an extended evaluator. Unfortunately, each of these extensions has been designed and implemented separately and is hardwired into a particular attribute grammar system. For potential users, adding new abstractions is non-trivial, since it requires modification of the attribute evaluation system itself. For example, it can sometimes be useful to thread attribute values from right-to-left (e.g., when computing backward slices or use-def relations between variables). In a system with only left-to-right chained attributes, this dependence must be encoded using basic attribute equations, despite the similarity of the abstractions.

In his OOPSLA'98 invited talk, "Growing a Language" [29], Guy Steele argued that "languages need to put the tools for language growth in the hands of the users," providing high-level language features that abstract over various extensions, rather than directly providing language features to solve specific problems. To this effect, we propose *attribute decorators* as a solution for the extensibility problem of attribute grammar specification languages. A decorator is a generic declarative description of the tree traversal or evaluation mechanism used to determine the value of an attribute. Decorators augment basic attribute equations with additional behavior, and can provide non-local dependencies or a form of search as required. For instance, a decorator can specify that the value of an attribute is to be sought at the parent node (and recursively higher in the tree) if it is not defined at the current node. Decorators can also enhance the usability of attribute equations for specific domains, separating the generic behavior from specific equations such as type checker constraints or data-flow equations, supported in other systems through specialized extensions.

In this paper, we present ASTER, a system for *decorated attribute grammars* (available from [1]). Decorators are powerful enough to specify all of the attribute grammar extensions listed above, avoiding the need to hardwire these into the system. A library of decorators suffices to cover common cases, while user-defined, domain-specific decorators can be used for specific applications and domains.

Decorators are inspired by *strategic programming*, where generic traversal strategies enable a separation between basic rewrite rules defining a tree transformation and the way in which they are applied to a tree [35,21,22]. In our case, local attribute equations define the core values of a tree computation, while decorators describe how those values are combined across the tree structure. The ASTER specification language is built as an extension of the Stratego strategic programming language [9]. We reuse the generic traversal operators of Stratego for the specification of decorators, and its pattern matching and building operations as the basis for attribute equations.

We begin this paper with background on attribute grammars and introducing our basic notations. Section 3 defines decorators, showing how they augment basic equations and capture common patterns. In Section 4 we present typical language engineering applications, demonstrating how decorators can be effectively applied in this area. We briefly outline our implementation in Section 5. We conclude with a comparison to related work and pointers to future directions. In this extended version of the paper, Appendix A and B elaborate on data-flow analysis and the implementation of ASTER.

```
1  eq Root(t):
2    t.global-min := t.min
3    id.min       := t.min
4    id.replace   := Root(t.replace)

5  eq Fork(t1,t2):
6    t1.global-min := id.global-min
7    t2.global-min := id.global-min
8    id.min     := <min> (t1.min,t2.min)
9    id.replace := Fork(t1.replace,
10                      t2.replace)

11 eq Leaf(v):
12   id.min     := v
13   id.replace := Leaf(id.global-min)
```

Fig. 1. An attribute grammar specification for *repmin* in pattern major form.

```
eq min:
  Root(t)     → t.min
  Fork(t1,t2) → <min> (t1.min,t2.min)
  Leaf(v)     → v

eq global-min:
  Root(t).t      → id.min
  Fork(t1,t2).t1 → id.global-min
  Fork(t1,t2).t2 → id.global-min

eq replace:
  Root(t)     → Root(t.replace)
  Fork(t1,t2) → Fork(t1.replace,
                     t2.replace)
  Leaf(v)     → Leaf(id.global-min)
```

Fig. 2. An attribute grammar specification for *repmin* in attribute major form.

2 Attribute Grammars

As they were originally conceived, attribute grammars (AGs) specify dependencies between attributes of adjacent tree nodes [20]. Attributes are generally associated with context-free grammar productions. For example, consider a production `X ::= Y Z`. Attribute equations for this production can define attributes for symbols `X`, `Y` and `Z`. Attributes of `X` defined at this production are called *synthesized*, as they are defined in the context of `X`. They can be used to pass information upwards. Conversely, attributes of `Y` and `Z` defined in this context can be used to pass information downwards, and are called *inherited* attributes.

2.1 Pattern-Based Attribute Grammars

In this paper we adopt a notational variation on traditional AGs in which attribute equations are associated with tree or term patterns instead of grammar productions [12,7]. Trees can be denoted with prefix constructor *terms* such as `Root(Fork(Leaf(1),Leaf(2)))`. Tree patterns for matching and construction are terms with variables (indicated in italics throughout this paper), e.g. `Fork(`t_1`,`t_2`)`.

Basic attribute equations have the form

`eq` p`:` r`.`a `:=` v

and define equations for a term that matches pattern p, where attribute a with a relation r to the pattern has value v. The relation r can be a subterm of p indicated by a variable or the term matched by the pattern itself, indicated by the keyword `id`.

As an example, consider the transformation known as Bird's *repmin* problem [4], which can be well expressed as an AG, as illustrated in Figure 1. In this transformation, a binary tree with integer values in its leaves is taken as the input, and a new tree with the same structure and its leaves replaced with the minimum leaf value is produced as the output. For example, the tree `Root(Fork(Leaf(1),Leaf(2)))` is transformed to `Root(Fork(Leaf(1),Leaf(1)))`.

In the specification of Figure 1, the local minimum leaf value in a subtree is computed in the synthesized attribute `min` (lines 3, 8 and 12). At the top of the tree, the minimum for the whole tree is copied to the inherited `global-min` attribute (line 2), which

is then copied down the tree to the leaves (lines 6 and 7). Finally, the `replace` attribute constructs a tree where each leaf value is replaced by the global minimum (lines 4, 9, 13).

Attribute equations are often defined in sets that share a common pattern, but may also be grouped to define a common attribute, which can make it easier to show the flow of information at a glance. Consider Figure 2, which is equivalent to the specification in Figure 1, but organizes the equations per attribute instead. Equations can be defined in separate modules, across different files, and are automatically assembled into a complete specification. Thus, language definitions can be factored per language construct and/or per attribute to support modular, extensible language definitions [15,32].

Using patterns helps separation of concerns when specifying a syntax and AG analyses. However, it can still be useful to use the concrete syntax of a language. ASTER supports this using the generic approach of *concrete object syntax embedding* as described in [34]. For example, instead of a pattern `While(`e`,`s`)`, we can use a concrete syntax pattern, which is typically enclosed in "semantic braces":

```
eq |[ while (e) s ]|:
  id.condition = e
```

Concrete syntax patterns are parsed at compile-time, and converted to their abstract syntax equivalents. Section 4 includes further examples of this technique.

2.2 Copy Rules

In theory, basic attribute equations with local dependencies are sufficient to specify all non-local dependencies. Non-local dependencies can be encoded by passing context information around using local inherited and synthesized attributes. In the repmin example, this pattern can be seen in the definition of the global minimum value, which is defined in the root of the tree. This information is passed down by means of so-called *copy rules*, equations whose only purpose is to copy a value from one node to another.

To accommodate for the oft-occurring pattern of copying values through the tree, many AG systems provide a way to *broadcast* values, eliminating the need for tedious and potentially error-prone specification of copy rules by hand. For example, the repmin example can be simplified using the `including` construct of the GAG and LIGA systems [17], which provide a shorthand for specifying copy rules. Using this construct, the copy rules in Figure 1, lines 6 and 7 could be removed and line 13 replaced by `id.replace := Leaf(including Root.global-min)`, specifying that the value is to be copied downward from the `Root` node.

3 Decorators

While constructs such as `including` provide notational advantages for some specifications, they cannot be used if the desired pattern of attribution does not precisely fit their definition. These notations are built into AG systems, and as such a developer is faced with an all-or-nothing situation: use a nice abstract notation if it fits exactly or fall back to writing verbose copy rules if there is no suitable shorthand. This section proposes attribute decorators as a more flexible alternative to building these shorthand abstractions into the AG system. Decorators can be defined to specify how attribute values are to be

propagated through the tree. Common patterns such as `including` can be provided in a decorator library, while user-defined decorators can be written for other cases.

To define high-level attribute propagation patterns, we draw inspiration from strategic programming [35,21,22]. This technique allows the specification of traversal patterns in a generic fashion, independent of the structure of a particular tree, using a number of basic, generic traversal operations.

3.1 Basic Attribute Propagation Operations

Consider the specification of Figure 3. It specifies only the principal repmin equations, avoiding the copy rules. The flow of information is instead specified using decorators (at the top of the specification). For instance, `global-min` uses the `down` decorator, which specifies that values should be copied downwards. Before we elaborate on the decorators used in this example, let us first examine the unabbreviated set of equations and reduce them to a more generic form that uses elementary propagation operations. After this, we will show how these operations can be used in the specification of decorators.

Downward propagation of the `global-min` attribute, first defined at the root of the tree (as seen in Figure 3), was originally achieved by

```
eq Fork(t1,t2):
  t1.global-min := id.global-min
  t2.global-min := id.global-min
```

Another reading of this specification says that 'the `global-min` of any non-root term is the `global-min` of its parent.' Thus, if we can reflect over the tree structure to obtain the *parent* of a node, we can express this propagation as

```
eq Fork(t1,t2):
  id.global-min := id.parent.global-min
eq Leaf(v):
  id.global-min := id.parent.global-min
```

```
def down        global-min
def up          min
def rewrite-bu replace

eq Root(t):
  t.global-min := id.min

eq Fork(t1,t2):
  id.min := <min> (t1.min,
                   t2.min)
eq Leaf(v):
  id.min     := v
  id.replace :=
     Leaf(id.global-min)
```

Fig. 3. Repmin using decorators.

This notation makes the relation to the parent node's attribute value explicit, rather than being than implied by the context. It forms the basis of specifying the downward propagation in a more generic way: `id.parent.global-min` could be used as the default definition of `global-min`, used for nodes where no other definition is given (here, all non-root nodes). This is essentially what the `down` decorator in Figure 3 does.

A different form of propagation of values was used in the `replace` attribute:

```
eq replace:
  Root(t)     → Root(t.replace)
  Fork(t1,t2) → Fork(t1.replace, t2.replace)
```

Here we can recognize a (common) rewriting pattern where the node names remain unchanged and all children are replaced. We abstract over this using the `all` operator:

```
eq replace:
  Root(t)     → all(id.replace)
  Fork(t1,t2) → all(id.replace)
```

`all` is one of the canonical *generic traversal operators* of strategic programming [35,21]. It applies a function to all children of a term. Other generic traversal operators include `one`, which applies a function to exactly one child, and `some`, which applies it to one or

more children. In this case, we pass `all` a reference to the `replace` attribute. This reveals an essential property of attribute references in ASTER: they are *first-class citizens* that can be passed as the argument of a function in the form of a closure. The expression `id.replace` is a shorthand for a closure of the form $\lambda t \rightarrow (t.\texttt{replace})$. It can be applied to the current term in the context of an attribute equation or in a sequence of operations, or to a term t using the notation <f> t.

3.2 Attribute Propagation using Decorators

We implement attribute definitions using functions that map terms to values. Parts of such a function are defined by attribute equations. Some attribute definitions form only a partial function, such as those in Figure 3. In that figure, copy rules are implicitly provided using decorators. Decorators are essentially higher-order functions: they are a special class of attributes that take another attribute definition (i.e., function) as their argument, forming a new definition with added functionality. This means that the declaration `def down global-min` and the accompanying equations for the `global-min` attribute effectively correspond to a direct (function) call to decorator `down`:

```
eq Root(t):
  t.global-min := id.down(the original global-min equations, here t.min)
```

A basic decorator d decorating an attribute a is specified as follows:

```
decorator d(a) = s
```

The body s of a decorator is its evaluation strategy, based on the Stratego language [9]. It provides standard conditional and sequencing operations. Using *generic traversal operators*, the evaluation strategy can inspect the current subtree. These operators are agnostic of the particular syntax used, making decorator definitions reusable for different languages. In this paper, we introduce the notion of *parent references* as an additional generic traversal operator, in the form of the `parent` attribute. Furthermore, we provide a number of *generic tree access attributes* that are defined using these primitives, such as the `prev-sibling` and `next-sibling` attributes to get a node's siblings, and `child(`c`)` that gets the first child where a condition c applies. Finally, we introduce *reflective attributes* that provide information about the attribute being decorated. These include the `defined` attribute, to test if an attribute equation is defined for a given term, and the `name` and `signature` attributes to retrieve the attribute's name and signature.

To illustrate these operations, consider the definition of the `down` decorator, which defines downward propagation of values in the tree (see Figure 4). This decorator automatically copies values downwards if there is no attribute equation defined for a given node. It checks for this condition by means of the `defined` reflective attribute (1). In case there is a matching equation, it is simply evaluated (2). Otherwise, the decorator acts as a copy rule: it "copies" the value of the parent. For this it recursively continues evaluation at the parent node (3). Conversely, the `up` decorator provides upward propagation of values. If there is no definition for a particular node, it inspects the child nodes, eventually returning the first successful value of a descendant node's attribute equation.

The `rewrite-bu` decorator provides bottom-up rewriting of trees, as we did with the `replace` attribute. Using the `all` operator, it recursively applies all defined equations for an attribute, starting at the bottom of the tree. Rewrites of this type produce a

```
  decorator down(a) =
(1) if a.defined then
(2)   a
    else
(3)   id.parent.down(a)
    end

  decorator up(a) =
    if a.defined then
      a
    else
      id.child(id.up(a))
    end
```

```
decorator rewrite-bu(a) =
  all(id.rewrite-bu(a))
; if a.defined then
    a
  end

decorator down at-root(a) =
  if not(id.parent) then
    a
  else
    fail
  end
```

Fig. 4. Basic decorator definitions.

new tree from an attribute, which in turn has attributes of its own, potentially allowing for staged or repeated rewrites.

In the following sections we provide some examples of more advanced decorators. At their most elaborate, these may specify a pattern p, can be parameterized with functions $a*$ and values $v*$, and may themselves be decorated ($d*$):

```
decorator d* [p .] name (d [, a* ] [| v* ]) = s
```

Note in particular the vertical bar '|', used to distinguish function and value arguments; in a call `f(|`x`)`, x is a value argument, in a call `f(`x`)` it is a function. The same convention, based on the Stratego notation, is supported for attributes. Furthermore, note that decorators can import other decorators $d*$. Such decorators are said to be *stacked*, and provide opportunity for reuse. To illustrate this, consider the `at-root` decorator of Figure 4. It evaluates attribute equations at the root of a tree, where the current node has no parent. Using the `down` decorator, the result is propagated downwards. Effectively, applying this stacked decorator results in a function application of the form `id.down(id.at-root(`a`))`. Stacking can also be achieved by declaring multiple decorators for an attribute. For example, we can add a "tracing" decorator to the `global-min` attribute, logging all nodes traversed by the `down` decorator:

```
def down trace global-min
```

4 Applications

In this section we discuss a number of common idioms in AG specifications, and show how attribute decorators can be used to encapsulate them. We focus on language processing, a common application area of AG systems. As a running example we use a simple "while" language (see Figure 5). We demonstrate different language analysis problems and how they can be dealt with using high-level decorators that are agnostic of the object language. As such, they are reusable for more sophisticated languages and other applications.

4.1 Constraints and Error Reporting

A fundamental aspect of any language processing system is reporting errors and warnings. We define these as declarative *constraints* using conditional attribute equations.

```
Program ::= Function*                    Type    ::= IntType | ...
Function ::= function ID(Arg*) { Stm* }  IntType ::= int
Stm     ::= { Stm* }                     Arg     ::= ID : Type
          | if (Expr) Stm else Stm       Expr    ::= Int | Var | ID(Expr*)
          | while (Expr) Stm                       | Expr + Expr | Expr * Expr
          | var ID : Type | ID := Expr   Int     ::= INT
          | return Expr                  Var     ::= ID
```

Fig. 5. The "while" language used in our examples.

These equations specify a pattern and a conditional `where` clause that further restricts the condition under which they successfully apply:

```
eq error:
  |[ while (e) s ]| → "Condition must be of type Boolean"
  where not(e.type ⇒ BoolType)

  |[ e1 + e2 ]|     → "Operands be of type Int"
  where not(e1.type ⇒IntType; e2.type ⇒IntType)
```

Each equation produces a single error message string if the subexpression types do not match `IntType` or `BoolType`. Rather than having them directly construct a list, we can collect all messages using the `collect-all` decorator (see Figure 6). It traverses the tree through recursion, producing a list of all nodes where the attribute succeeds. Note that this decorator does not test for *definedness* of the equations (using a`.defined`), but rather whether they can be successfully applied. Using `collect-all` with the `error` attribute, we can define a new `errors` attribute:

```
def collect-all errors :=
  id.error
```

This notation both declares the decorators and a default equation body, which refers to `error`.

```
decorator node.collect-all(a) =
  let results =
    node.children.map(id.collect-all(a))
  ; concat
  in if <a> node then // add to results
       ![<a> node | <results>]
     else
       results
     end
  end
```

Fig. 6. The `collect-all` decorator.

To provide usable error messages, however, the error strings need further context information. We can define a new, application-specific decorator to add this information before they are collected, and use it to augment the `error` attribute:

```
decorator add-error-context(a) =
  <conc-strings> (a," at ",id.pp," in ",id.file,":",id.linenumber)

def add-error-context error
```

With this addition, the `errors` attribute now lists all errors, including a pretty-printed version of the offending construct (provided a `pp` attribute is defined), and its location in the source code (given a `file` and `linenumber` attribute).

4.2 Name and Type Analysis

Type analysis forms the basis of static error checking, and often also plays a role in code generation, e.g. for overloading resolution. Types of expressions typically depend on local operands, or are constant, making them well-suited for attribute equations. Moreover, an AG specification of a type analysis is highly modular, and may be defined

across multiple files. Thus, let us proceed by defining a `type` attribute for all expressions in our language to perform this analysis:

```
eq type:
  Int(i)          → IntType
  |[ e1 + e2 ]|   → IntType where e1.type⇒IntType; e2.type⇒IntType
  Var(v)          → id.lookup-local(|v).type
  |[ f(args) ]|   → id.lookup-function(|f, args).type
```

Variable references and function calls require non-local *name analysis* to be typed. This can be done using parameterized *lookup attributes* that given a name (and any arguments), look up a declaration in the current scope [14]. In the example we reference the local `type` attribute of the retrieved node, but lookup attributes can be used to access arbitrary non-local attributes for use in various aspects of the system. The actual lookup mechanism is provided by means of reusable decorators: to do this for a particular language, it suffices to select an appropriate decorator and define the declaration sites and scoping constructs of the language. Our lookup attributes are defined as follows:

```
def lookup-ordered(id.is-scope) lookup-local(x) :=
  id.decl(|x)

def lookup-unordered(id.is-scope) lookup-function(|x, args) :=
  id.decl(|x, args)
```

Figure 7 shows the prerequisite `decl` and `is-scope` attribute definitions for the name analysis, specified as arguments of the above attributes. Again, these are highly declarative and each address a single aspect. Declaration sites are identified by the `decl` attribute, which is parameterized with an identifier name x and optionally a list of arguments. It only succeeds for matching declarations. All declarations also define a `type` attribute. Similarly, the `is-scope` attribute is used to identify scoping structures. Note in particular the equations of the "if" construct, which, for the purpose of this example, defines scopes for both arms, similar to try/catch in other languages.

Languages employ varying styles of scoping rules. In our language we have two kinds of scoping rules: C-like, *ordered* scoping rules, and Algol-like, *unordered* scoping rules. In many languages, local variables typically use the former, while functions typically use the latter. We define the `lookup-ordered` and `lookup-unordered` dec-

```
eq |[ var x : t ]|:
  id.type     := t
  id.decl(|x) := id

eq |[ x : t ]|: // function parameters
  id.type     := t
  id.decl(|x) := id

eq |[ function f(params) : t stm ]|:
  id.type          := t
  id.decl(|f, args) := id where params.map(id.type).eq(|args.map(id.type))

eq is-scope:
  |[ function f(params) : t { stm* } ]| → id
  |[ if (e) s1 else s2 ]|.s1 → s1
  |[ if (e) s1 else s2 ]|.s2 → s2
  |[ while (e) s ]|          → id
  |[ { s* } ]|               → id
```

Fig. 7. Attributes for name analysis and types of declarations.

```
  decorator down lookup-ordered(fetch-decl, is-scope) =
(1)  fetch-decl
(2)  <+ id.prev-sibling(lookup-outside-scopes(fetch-decl, is-scope))

  decorator down lookup-unordered(fetch-decl, is-scope) =
    (id.is-root <+ is-scope) // only look in scoping structures
(4)  ; lookup-in-scope(fetch-decl, is-scope)

  lookup-in-scope(fetch-decl, is-scope) =
    fetch-decl
    <+ id.child(lookup-outside-scopes(fetch-decl, is-scope)) // enter scope

  lookup-outside-scopes(fetch-decl, is-scope) =
    fetch-decl
(3)  <+ not(is-scope) // do not enter scope subtrees
     ; id.child(lookup-outside-scopes(fetch-decl, is-scope))
```

Fig. 8. Lookup attributes and decorators.

orators to accommodate for these styles (see Figure 8). They traverse up the tree, inheriting the behavior of the `down` decorator, thus giving precedence to innermost scopes. Along this path, the `lookup-ordered` decorator visits the current node (1). If no declaration is found there (i.e., `fetch-decl` fails), the `<+` combinator specifies that it should proceed at (2), visiting any preceding siblings using the helper function `lookup-outside-scopes`. This function performs a local lookup for declarations in these nodes, respecting the scoping rules by avoiding traversal of scoping constructs (3). In contrast, `lookup- unordered` follows a straight path to the root of the tree, doing a search in encountered scopes (4).

4.3 Flow Analysis

Control-flow analysis forms the foundation of data-flow analysis, which is prerequisite to various compiler optimizations, refactorings, and static checks for bug patterns or security violations. A recent paper by Nilsson-Nyman et al. [26] demonstrated how AGs can be employed for modularly specifying such analyses, ensuring separation of concerns and reusability with different data-flow analyses.

We take an approach similar to that of the JastAdd project, using reference attributes [14] to declaratively define the control flow graph. Consider Figure 9, which defines a `succ` attribute, providing a reference to all successors of a statement. For instance, for the "if" statement, the successors are the "then" and "else" branches (1). A helper attribute, `succ-enclosing`, determines the default successors based on the enclosing block. For sequences of statements, the successor is the next statement in the sequence (2). The "while" statement overrides this behavior, by setting the successor of the enclosed block to

```
  def down succ-enclosing:
    Program(_)              → []
(2) [s1, s2 | _].s1         → [s2]
(3) |[ while (e) s ]|.s     → [id]

(4)def default(id.succ-enclosing) succ:
    |[ { s; s* } ]|         → [s]
(1) |[ if (e) s1 else s2 ]| → [s1, s2]
    |[ return e ]|          → []
    |[ while (e) s ]|
        → [s|s.succ-enclosing]

(5)decorator default(a, default) =
    if a then
      a
    else
      default
    end
```

Fig. 9. Specification of the control flow.

```
  decorator contributes-to(a, targets) =
    if not(completed survey phase) then
(1)   mark survey phase complete
    ; id.root
    ; in a topdown fashion:
        for a node x, apply targets and add them to the list of contributions for x
  ; end
(2); apply a to the list of contributions for the current node
```

Fig. 10. The `contributes-to` decorator, contributing values to a list of nodes.

itself (3). For any non-control flow statements, we specify `succ-enclosing` as the default successor `succ` (4), using the `default` decorator (5).

The specification of the `succ` attribute allows for a natural, declarative way of specifying the forward control flow of a language. However, a number of data-flow analyses depend on the *predecessors* of a statement. To avoid specifying these by hand, it is possible to use *collection attributes* [6,23,5] to *derive* the reverse flow graph. Collection attributes introduce a *"contributes to"* clause, allowing nodes to contribute values to collections in other nodes. Using this technique, we can define the predecessor graph in a single equation, by contributing each statement to its successors:

```
def contributes-to(id.succ) stm:
  id.pred := stm
```

Figure 10 defines the `contributes-to` decorator. Note that for clarity, we use fragments of pseudocode in lieu of more advanced Stratego constructs. The complete, 20-line source is available from [1]. This decorator operates in two phases: the first time any collection attribute is evaluated, it enters the *survey phase* (1), where the complete tree is traversed, adding all contributing nodes to a list maintained for each node contributed to. This is done only once, rather than for every collection attribute retrieved. After this phase completes (2), referenced collections only require the application of any attribute equations associated with it (for `pred`, `stm` is returned). Note that all required bookkeeping operations (i.e., storing contributions and whether the survey phase completed) are performed in the context of the current attribute: they are stored in tables associated with the attribute's unique signature and its argument values (i.e., `id.signature`).

The control flow graph, specified by the `succ` and `pred` attributes, forms the foundation of any data-flow analysis. As such a graph may have cycles in it, these analyses have the peculiar property that their equations may involve circular dependencies. This makes them unsuitable for traditional AGs. However, by extending the formalism with *circular attributes* [24,5], it becomes possible to use declarative AG equations to specify such analyses [26]. Circular attribute equations can be solved by fixed point iteration, as long as their underlying data forms a lattice. We implemented this in a decorator that evaluates circular attributes. However, due to the relatively intricate nature of the algorithm that underlies this decorator, we do not include this in the body of this paper, but rather refer the interested reader to Appendix A.

5 Implementation

The ASTER language is built as an extension of the Stratego strategic programming language [9], which natively supports the canonical generic traversal operators. The

ASTER compiler is implemented in standard Stratego, using only a (bootstrapped) AG specification for error reporting (using constraint rules similar to those in Section 4.1). It compiles AG specifications to regular Stratego programs through a series of normalization steps. The normalization process starts by grouping attribute equations together, forming separate strategies for each attribute and decorator. As illustrated in Section 3.2, attribute equations and decorators are implemented as functions with generic traversal operations (called *strategies* in strategic programming). Inherited attributes are defined at the parent of a node; therefore, their implementation uses the `parent` primitive. Attribute references and imported decorators are converted to strategy calls. For decorator calls, static reflective data is added for reflective attributes such as `signature`. Finally, a memoization mechanism is added to cache all attribute and decorator calls. In Appendix B we elaborate on the different normalization steps, using the repmin specification as an example.

Using memoization, attributes are evaluated at most once, thus achieving optimal evaluation. Similar memoization-based dynamic evaluation has been used before in many other systems, e.g. by Jalili [16] and recently in JastAdd [15]. In ASTER, memoization can be selectively disabled and overridden with custom behavior using decorators. For example, we disable it for the data-flow analysis of Appendix A.

Our current, experimental implementation has not been tuned for performance. One constraining factor is currently the ATerm library used to represent trees, which forms an integral part of Stratego. It is optimized for a maximally shared representation of terms, where identical subtrees occupy the same space in memory [8]. This makes it less suitable for storing additional, dynamic information in tree nodes, in our case parent references (for `id.parent`) and memoized attribute values. We worked around this by annotating tree nodes with unique keys, and use these to store the added information in separate tables. In the future, we would like to adapt or replace the underlying implementation to better accommodate for this. Regardless, preliminary performance measurements indicate promising results. We compared our compiler against JastAdd [15], a mature AG system that uses an evaluation mechanism conceptually very similar to our own. We used the repmin program of Figure 1 as a test case. Over an average of fifty runs, JastAdd took 51 ms to replace all leaves in a large tree with 2^{16} leaves. Our system took 150 ms, or 180 ms for the version of Figure 3 where decorators are used in place of manual copy rules. Further testing confirms an unfortunate, but constant overhead of about a factor three in the base performance level, due to the expensive memoization and term initialization operations. Still, the results indicate a low overhead of the decorator mechanism. Furthermore, both our specifications, especially when using decorators, are more concise than the version implemented in JastAdd.

6 Related Work

The general principle behind attribute decorators shares similarities with the Decorator design pattern, which describes how to add functionality to objects at run-time [13]. Variations of this idea exist in languages such as Python, which features decorators for functions [28]. In our case, we augment basic attribute definitions with either propagation of values from other nodes or with higher-level behavior such as a circular

evaluation scheme. This kind of augmentation is similar to code weaving used in many forms of aspect-oriented programming [19].

Although considerable research has been devoted to various special-purpose extensions of AGs (as illustrated in the preceding sections), rather less attention has been paid to extensibility of AG systems. Two systems that do aim at different degrees of extensibility are Silver [30] and first-class attribute grammars [25].

In first-class AGs, attribute equations are *first-class citizens*, allowing them to be combined and manipulated using the language itself. Using function combinators, basic basic up, down, and chain copy rules can be defined [25]. These combinators show similarities with decorators, although they are purely defined in terms of functional dependencies, and lack the reflective and traversal primitives that form the building blocks of decorators. The paper does not indicate that they could be used to implement more sophisticated forms of propagation and manipulation of equations, such as the collection and circularity decorators. Based on the Haskell type checker, first-class AGs prevent errors where the use of an attribute does not match its type. Errors due to cyclic dependencies or a mismatch between attribute equations and grammar productions are not reported. Our system is based on Stratego, which is largely untyped (but could be typed [22]). Further complicated by the use of parent node references, it currently does not provide a fully typed system, other than basic static pattern coverage checking.

Silver supports extension with automatic copy rules as well as more advanced features such as collection attributes in a relatively accessible manner [30]. Implemented in itself, the Silver language can be used to modularly implement such extensions. While adding extensions of this kind is made easier through facilities such as forwarding for local transformations [31] and higher-order attributes, it is hard to imagine a regular Silver user building such an extension. Moreover, it is difficult to encapsulate these extensions in a single application or library, as they must be integrated in the base AG system. In contrast, many decorators are light-weight so they can be developed quickly and easily as needed.

A system that particularly inspired our design has been JastAdd [15], which extends traditional AGs in a number of interesting ways.[1] JastAdd uses reference attributes [14], which we also use in a number of decorators. Its extensions include collection attributes [23] and circular computations [24]. These are built into the JastAdd implementation; there is no user-level mechanism to define similar extensions. As demonstrated in Section 4, decorators can be used to define these same features at a higher level. Admittedly, we would not expect users to define relatively complex features like this very often, but building on the high-level framework provided by decorators is likely to be much easier than modifying the underlying implementation of an AG evaluation system. JastAdd is designed to be used in conjunction with hand-written code, particularly using visitors. As such, it provides a way to write traversals that interoperate with declarative attribution. In theory, this facility could be used to implement something similar to decorators, but this would require the addition of generic traversal on top of the Java implementation of trees, essentially duplicating the Stratego platform we use.

[1] For the purposes of this paper, we focus on the attribute grammar features of JastAdd, ignoring its support for rewriting trees during evaluation [11].

7 Conclusions and Future Work

We propose decorated attribute grammars as a formalism for application-level extensibility of AG systems. To this end, we have identified primitives for the specification of decorators to define abstract evaluation strategies for attributes. By means of a prototype implementation and by employing decorators in different language engineering applications, we demonstrated the feasibility of using decorators to implement common abstractions over basic attribute grammars. These can be provided in the form of a library, and may be extended with user-defined decorators, where decorator stacking can be applied to reuse existing definitions.

In the future, we would like to explore further applications of decorated attribute grammars, in particular in the domain of implementing domain-specific languages and modular language extensions. For this we want to build upon the rewriting capabilities of the Stratego transformation language, the foundation of ASTER. As such, we aim to take the best of both worlds; rewriting with Stratego and declarative analysis with attribute grammars.

Building on our past experience [18], another application area to which we want to apply ASTER is that of integrated development environments (IDEs). ASTER's performance is already sufficient to be usable, and its demand-driven evaluation further helps interactive application. As such, we would like to employ it as part of an IDE in the future, encapsulating logic for typical editor service components, incremental compilation concerns, and related patterns in decorators.

Acknowledgments We would like to thank Nicolas Pierron for the discussions on attribute grammar systems and their implementation. This research was supported by NWO projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, *TFA: Transformations for Abstractions*, and 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*.

References

1. Aster project home page. http://strategoxt.org/Stratego/Aster.
2. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
3. A. Baars, D. Swierstra, and A. Löh. UU AG System User Manual. *Department of Computer Science, Utrecht University, September*, 2003.
4. R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.
5. J. Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, 1996.
6. J. Boyland. Remote attribute grammars. *Journal of the ACM (JACM)*, 52(4):627–687, 2005.
7. J. Boyland and S. L. Graham. Composing tree attributions. In *POPL'94*, pages 375–388. ACM, 1994.
8. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
9. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

10. M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.
11. T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.
12. C. Farnum. Pattern-based tree attribution. In *POPL'92*, pages 211–222, 1992.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
14. G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.
15. G. Hedin and E. Magnusson. JastAdd – an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
16. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices*, 18(9):35–44, 1983.
17. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.
18. L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. Integrating SGLR into IMP. In *LDTA 2008*, April 2008.
19. G. Kiczales et al. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
20. D. E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.
21. R. Laemmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *Proceedings of Aspect-Oriented Software Development (AOSD 2003)*, pages 168–177, Boston, USA, March 2003. ACM Press.
22. R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54(1):1–64, 2003.
23. E. Magnusson, T. Ekman, and G. Hedin. Extending attribute grammars with collection attributes – evaluation and applications. *Proc. of the Int. Working Conference on Source Code Analysis and Manipulation*, pages 69–80, 2007.
24. E. Magnusson and G. Hedin. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.
25. O. de Moor, K. Backhouse, and S. Swierstra. First-class attribute grammars. *Informatica*, 24(3):329–341, 2000.
26. E. Nilsson-Nyman, T. Ekman, G. Hedin, and E. Magnusson. Declarative intraprocedural flow analysis of Java source code. In *LDTA 2008*, 2008.
27. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995.
28. G. van Rossum. *Python Reference Manual*. iUniverse, 2000.
29. G. Steele. Growing a language. *Higher Order Symb. Comp.*, 12(3):221–236, 1999.
30. E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. In *LDTA 2007*, volume 203 of *ENTCS*, pages 103–116. Elsevier Science, 2008.
31. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC'02*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.
32. E. Van Wyk, L. Krishnan, D. Bodin, and E. Johnson. Adding domain-specific and general purpose language features to Java with the Java language extender. In *Companion to OOPSLA'06*, pages 728–729. ACM, 2006.
33. E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for java. In *European Conference on Object Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer Verlag, July 2007.
34. E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, 2002.

35. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP 1998)*, pages 13–26. ACM, 1998.
36. H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Programming Language Design and Implementation (PLDI'89)*, pages 131–145. ACM Press, 1989.

A Data-flow Analysis Using Circular Attribute Equations

In this appendix we demonstrate how attribute decorators can be employed to specify a data-flow analysis, using attribute equations that involve circular dependencies. In Section 4.3 we have shown how a control flow graph can be defined using attribute equations, aided by the `default` and `down` decorators to form the `succ` successor attribute (see Figure 9). Using the `contributes-to` decorator, we defined the *reverse* control flow graph in the form of the `pred` attribute. We use it for the analysis presented here.

A typical data-flow analysis is that of *reaching definitions*, which forms the basis for a constant propagation optimization. Its definition is as follows [2]:

$$RD_{out}(b_i) = gen(b_i) \cup (RD_{in}(b_i) \backslash kill(b_i))$$
$$RD_{in}(b_i) = \bigcup_{x \in pred(b_i)} RD_{out}(x)$$

These equations define the "out" and "in" sets of *definitions* for each statement b_i, respectively defining all variable definitions that reach just after and just before a statement. At the beginning of each statement, the definitions that reach it are defined by the union of all reaching definitions of the predecessor statements $pred$. The "out" set is maintained by means of a so-called "gen" set and a "kill" set that indicate for each statement b_i the statements to be respectively added to it and removed from it. In our following example, we use these sets as a basis for the propagation of constant definitions: statements may *add* or *remove* (i.e., invalidate) constant definitions. For example, an assignment statement `x := 3` *adds* a new constant definition for variable x when it completes. Therefore, this definition is added to the "out" set of the statement.

While there is an obvious correspondence between the equations above and regular attribute equations, a typical characteristic of data-flow analyses is that their definition involves *circular dependencies*: the equations are defined in terms of each other. Given a control flow graph that has cycles in it, a traditional AG evaluator cannot solve these equations. Instead, an iterative process is required. Nilsson-Nyman et al. recently demonstrated how such equations can be naturally implemented for a liveness analysis in an attribute grammar system [26], by extension with *circular attributes* [24,5]. They implemented the extension itself in Java, as part of the JastAdd evaluator [24]. Another AG system that supports data-flow analysis is Silver [30]. However, rather than extending the AG evaluation system, it uses an external model checking tool to perform the analysis [33].

We provide a mechanism for circular attributes as part of the attribute grammar specification (or a library) rather than its evaluation system, in the form of the `circular` decorator. Instead of using the standard memoized evaluation scheme for attributes (see Section 5), this decorator provides an alternative evaluation strategy based on fixpoint iteration.

Consider Figure 11, which defines a constant propagation analysis using circular attributes and the `circular` decorator. In this definition, the `constant-out` and `constant-in` equations $(1, 2)$ correspond to the "out" and "in" sets seen before. Note again the dependency between the two equations. The `circular` decorator is declared for both attributes, where the empty list `[]` is the initial value. The constants removed

```
  def circular(|[]) stm:
(1) id.constant-out :=
      <id.table-union> (
        id.constant-genset,
        <id.table-diff> (id.constant-in, id.constant-killset)
      )

(2) id.constant-in := <id.isect> id.pred.map(id.constant-out)

(3) def default(![]) |[ var : t := e ]|:
      id.constant-genset  := [(var, e.constant-value)]
      id.constant-killset := [(var, ())] where not(e.constant-value)

(4) def constant-value:
      Int(i)         → i
      Var(v)         → id.constant-before.lookup(|v)
      |[ e1 * e2 ]|  → <add> (e1.constant-value, e2.constant-value)
      |[ e1 + e2 ]|  → <mul> (e1.constant-value, e2.constant-value)
```

Fig. 11. Constant propagation.

or added by each statement are defined by the "gen set" and "kill set" definitions (3). These sets are empty by default, but are specifically defined for assignments. In turn, these are defined by use of the by the `constant-value` attribute (4), which provides the constant value of an expression if one is currently available (or fails if an expression is not constant).

Following the basic algorithm of Magnusson et al. [24], the `circular` decorator we define here only allows a single fixpoint iteration to be active at a time. Any circular attributes on nodes referenced during its iteration are said to be its *participants*. Once the iteration has reached a fixpoint, all participants are marked `FINISHED`, and do not require a new fixpoint iteration. In practice, this means that for an intraprocedural analysis, a single fixpoint loop is performed per method. During the iterative process, any attribute encountered is marked `BUSY`, which indicates that it should not be recomputed until the next step of the iteration is entered.

Consider Figure 12, which defines the `circular` decorator and a number of helper functions. The main `circular` decorator determines how to evaluate an attribute based on its current state: for attributes marked `FINISHED` or `BUSY`, the current value is returned (1). If not (2), the `circular-fixpoint` helper (re)computes the attribute equation value (3). It also starts a new fixpoint iteration as required, which runs while any of the participants records a change (4).

To recompute the value of equations, the `recompute-circular-def` helper performs the required bookkeeping operations: any active equation is registered as `BUSY` (5), all participants of the iteration are added to a list (6), any changes to values are recorded (7), and the newly computed value is stored using the `put-cache` operation (8). Like for the `contributes-to` decorator of Section 4.3, all bookkeeping operations are performed in the context of the current attribute signature, with the exception of the global "fixpoint running" property.

Since the `circular` decorator evaluates equations multiple times, it is necessary to disable the default memoization behavior of equations (outlined in Section 5). We do this by importing the `plain` decorator, a built-in decorator that indicates that this behavior must be disabled for the decorator and any attributes that are transitively dec-

```
  def plain node.circular(a|initial) =
    (get-cache(|a.signature, node) <+ initial.init <+ !EVAL_FAILED())
  ; if ?FINISHED(value) + ?BUSY(value) then
(1)   !value // currently finished or busy; return last value
    else
(2)   fixpoint(a|a.signature, node, id)
    end
  ; not(EVAL_FAILED) // fail if failure placeholder encountered

  fixpoint(a|signature, node, oldvalue) =
(3) recompute(|signature, node, oldvalue)
  ; if not(fixpoint is running) then
      globally mark fixpoint running
(4) ; while marked changed:
        clear the old list of participants // eliminate stale participants
      ; apply recompute, with the latest result as its oldvalue argument
    ; mark all participants FINISHED
    ; globally unmark fixpoint running
    end

  recompute(a|signature, node, oldvalue) =
(5) node := <put-cache(|signature, node, BUSY(oldvalue))>
  ; (s <+ !EVAL_FAILED()) // evaluate or use placeholder for failure
  ; if id.eq(|oldvalue) then
(6)   register node as a participant
      put-cache(|signature, node, oldvalue) // no longer BUSY
    else
      id.init
(7) ; put-cache(|cache, node, id)
(8) ; globally mark changed
    end

  get-cache(|table, key) = retrieve a value from the cache
  put-cache(|table, key, value) = store a value in the cache
```

Fig. 12. The circular attribute evaluation decorator.

orated by it. Thus, it forms the basis for attributes equations that require a customized caching scheme. In the `circular` decorator, we use the `put-cache` and `get-cache` helper functions to control the standard caching operations. The `plain` decorator also disables automatic term initialization, i.e., annotating a term with unique keys (see Section 5). This allows us to do this in a more fine-grained fashion for this decorator, using the `init` attribute to explicitly initialize result terms as necessary. In the future, as we transition to an implementation that no longer uses these annotations, this operation should no longer be required.

In this paper we have shown how circular attributes can be implemented as a reusable library or application component, using strategic programming primitives to specify decorators, avoiding adaptation of the base AG system itself. For a more elaborate discussion of the uses of and evaluation algorithms behind circular attributes in general, we kindly refer the reader to [24] and [5].

B Normalization of Attribute Grammar Specifications to Stratego

ASTER specifications are compiled to Stratego programs through a series of normalization steps. The Stratego compiler [9] then compiles it to C. A Java-based back-end is under development. In this appendix, we illustrate the different normalization steps using the repmin specification of Section 3 as an example.[2]

B.1 Basic Attribute Equations

The original repmin attribute grammar was specified in pattern major form, i.e. groups of equations were together for a single pattern (as seen in Figure 3). In this form, attributes for different entities can be defined across multiple modules. Likewise, decorators can be declared separately from the equations of an attribute. The ASTER compiler parses this specification, and groups all attribute definitions together ensuring that attributes are defined at a single location:

```
def down global-min:
  Root(t) → id.min

def up min:
  Fork(t1, t2) → <min> (t1.min, t2.min)
  Leaf(v)      → v

def rewrite-bu replace:
  Leaf(v) → Leaf(id.global-min)
```

Any concrete syntax fragments in the specification are parsed and converted to their abstract syntax equivalents (as described in [34]).

Next, attribute equations are rewritten to regular Stratego rewrite rules. At this point, the mapping of equations to rewrite rules is relatively straightforward. For each attribute, a strategy is also generated, which governs the evaluation order of the rules and invokes the decorators declared for an attribute. For the `min` attribute, the following strategy and rewrite rules are generated:

```
def-min =
  id.up(eq-min-1 <+ eq-min-2)

eq-min-1:
  Leaf(v) → v

eq-min-2:
  Fork(t1, t2) → <min> (t1.min, t2.min)
```

The order in which the rewrite rules `eq-min-1` and `eq-min-2` are evaluated is determined not based on the lexical order of the equations, but instead based on their patterns. Each is assigned a priority determined by the number of constructors and literals in the pattern. For example, a pattern with a literal argument, such as `Leaf(3)`, has a higher priority than `Leaf(`v`)`. Equations with high-priority patterns are considered first for evaluation. This effectively means that more specific patterns generally override definitions that are not as specific. While it is not possible to do a full ordering of patterns

[2] Note that for the purpose of this paper, we introduce the required implementation details in more fine-grained steps than (and sometimes slightly deviating from) the current implementation at [1].

based on their specificity, our approach is deterministic and reasonably easy to follow, thus mostly avoiding unexpected orderings.

Inherited attributes such as `global-min` are defined using a pattern that matches against the parent node. When they are translated to Stratego rules, they use the `parent` primitive to perform this match operation:

```
eq-global-min-1:
  _ → id.min
  where id.parent ⇒ Root(t)
```

B.2 Failure of Attribute Equations

When an equation in ASTER matches a node, but fails to produce a result (because the subsequent operations fail), the implementation returns a special value `EVAL_FAILED`. To implement this, the equation rules are normalized to a more primitive form, using Stratego's '`?`' match and '`!`' build operators:

```
eq-min-1 =
  ?Leaf(v)
; (!v; id.init <+ !EVAL_FAILED())

eq-min-2 =
  ?Fork(t1, t2)
; (<min> (t1.min, t2.min); id.init <+ !EVAL_FAILED())
```

Each successfully matching rule returns either the resulting value of the equation or `EVAL_FAILED`. A successful result is initialized as needed using the `init` primitive (discussed in Section 5). By automatically initializing all resulting values of attribute equations, these can themselves be attributed, thereby supporting higher-order attributes [36].

If a rule returns `EVAL_FAILED`, fallthrough on failure by the `<+` operator in the attribute strategy is avoided. This is makes attributes more predictable and helps in the specification of exceptional cases. For example, given the constraint checking equations of Section 4.1, it is possible to add equations with more specific patterns to introduce exceptions to the existing rules. (In the constraint rules, failure indicates that no error should be reported.)

B.3 Attribute References and Memoization

All attribute and decorator references are normalized to strategy invocations. For example, the attribute reference t_1`.min` is implemented as follows:

```
<def-min> t1; not(?EVAL_FAILED())
```

That is, the `def-min` strategy is invoked on t_1, and the result is checked to ensure that the special `EVAL_FAILED` value is never exposed to the programmer.

Unlike normal Stratego strategies, all attribute equations are memoized (with the exception of attributes annotated with the special `plain` decorator). Memoization of attributes is performed at their definition site. For decorated attributes, each (stacked) decorator invocation is memoized. For example, for the `min` attribute, the following strategy is generated:

```
def-min =
  ?node
; let cached =
    get-cache(|Attribute("min", [], []), node)
  <+
    decorator-up(
      (eq-min-1 <+ eq-min-2)
    , cached
    )
  ; put-cache(|Attribute("min", [], []), node, id)
  in
    cached
  end
```

In this strategy, `cached` is a local strategy definition, which either returns the cached value of the attribute or calculates a new value and caches it. In the latter case, the `up` decorator is invoked, with the undecorated equations as the first argument. As the second argument, we it pass the `cached` strategy, used for memoization of recursive invocations of the attribute inside the decorator (i.e., `id.up(`a`)`).

B.4 Decorators

In Section 3 we defined the `up` decorator, used in the `min` attribute, as follows:

```
decorator up(a) =
  if a.defined then
    a
  else
    id.child(id.up(a))
  end
```

When translating this to a strategy, we add an additional parameter to ensure the attribute is properly memoized. We implement the `defined` primitive by evaluating the attribute: it is considered "defined" if it either evaluates successfully or returns `EVAL_FAILED`. The result is stored in a local variable. This approach ensures that both the pattern match operation and the remainder of the evaluation are evaluated only once. Below is the generated Stratego implementation of the `up` decorator:

```
decorator-up(a, cached) =
  if where(a-result := a) then
    !a-result
  else
    def-child(cached)
  end
<+
  !EVAL_FAILED()
```

Note in particular that the recursive call in this decorator is replaced by an invocation of `cached`, to ensure proper memoization. This call is passed as the argument of the parameterized `child` attribute, implemented by the `def-child` strategy.

B.5 Reflective and Administrative Data

The parent of each node in the tree is maintained in a table. Likewise, we maintain memoized values of each attribute in a table. Stratego has no true notion of global variables (it uses scoped dynamic rules instead [10]). Therefore, these tables are passed along as arguments of calls to attributes. Reflective data of attributes is also passed

along with decorator invocations. Adding these to the `def-min` strategy, the strategy controlling the evaluation of attribute `min` becomes:

```
def-min(|parent, all-parents, global-cache) =
  ?node
; let cached =
    get-cache(|Attribute("min", [], []), node)
  <+
    decorator-up(
      (  eq-min-1(|parent, all-parents, global-cache)
      <+ eq-min-2(|parent, all-parents, global-cache))
      ; initialize-attribute-value(|parent, all-parents, global-cache)
    , cached
    , parent
    , all-parents
    , global-cache
    , Attribute("min", [], [])
    )
  ; put-cache(|global-cache, Attribute("min", [], []), node, id)
  in
    cached
  end
```

In this example, the `all-parents` and `global-cache` parameters are the parent and memoization tables. These are passed to all attribute invocations. The `global-cache` variable is used with the `get-cache` and `put-cache` strategies. As a small optimization, we also pass the current node's parent (if known) as the `parent` parameter. Finally, the decorator invocation has an additional argument that specifies the signature of the decorated `min` attribute: `Attribute("min", [], [])`. We use the signature for manual caching operations (as seen in Appendix A) and for debugging, printing the name and arguments of an attribute. In the future, we would like to add further static reflective information of attribute equations, using it for more refined decorator definitions.

TUD-SERG-2008-038a
ISSN 1872-5392