Lightweight Language Processing in Kiama

Anthony M. Sloane

Department of Computing, Macquarie University, Sydney, Australia
Anthony.Sloane@mq.edu.au

Abstract. Kiama is a lightweight language processing library for the Scala programming language. It provides Scala programmers with embedded domain-specific languages for attribute grammars and strategy-based term rewriting. This paper provides an introduction to the use of Kiama to solve typical language processing problems by developing analysers and evaluators for a simply-typed lambda calculus. The embeddings of the attribute grammar and rewriting processing paradigms both rely on pattern matching from the base language and each add a simple functional interface that hides details such as attribute caching, circularity checking and strategy representation. The similarities between embeddings for the two processing paradigms show that they have more in common than is usually realised.

1 Introduction

Kiama is a language processing library for the Scala programming language [1, 2]. We are distilling the key ideas of successful processing paradigms from language research and making them available in a lightweight library. In other words, we are embedding the paradigms into a general purpose language. The result is a flexible combination of general programming techniques and high-level abstractions suited to many forms of language processing. At present, we are focused on building traditional language processing applications such as compilers, interpreters, generators and static analysis tools, but in the longer term we believe that these paradigms have much to offer in a more general software engineering setting.

Some of the motivation for Kiama comes from experience building generators for the Eli system [3]. Eli translates high-level specifications of language syntax, semantics and translation into C implementations. Eli successfully combines many off-the-shelf tools and custom-built generators that use a variety of specification languages. However, because of their varying origins, the Eli specification languages are often *ad hoc*, have arbitrary differences and lack features that are commonplace in general purpose languages such as name space control, modularity and parameterisation. In our view, as language processing systems are used to tackle larger tasks and different techniques are combined, these issues become particularly problematic.

The Kiama thesis is that in the language processing domain it is better to start with a modern general purpose language that embodies prevailing wisdom

about how to structure, scale and extend applications, than to expect every generator builder to incorporate this wisdom into their own specification languages and tools. The approach of Kiama is therefore to combine proven language processing paradigms into a coherent whole, supported by general facilities from a host language.

Kiama is hosted by the Scala programming language that provides both object-oriented and functional features in a statically-typed combination running on the Java Virtual Machine [1, 2]. Thus, Scala constitutes a powerful base on which to experiment with embedding. At present, Kiama supports two main processing paradigms: *attribute grammars* and *strategy-based term rewriting*. Attribute grammars are particularly suited to expressing computations on fixed tree or graph structures, which is needed for static analysis. Rewriting is ideal for describing computations that transform trees for translation or optimisation.

A notable result from our experience embedding attribute grammars and rewriting in Scala is the high degree to which the power of more complex generator-based systems can be realised with a lightweight embedding. For this domain at least, Scala provides just the right level of expressibility for the notations and flexibility for their implementation. Moreover, the two paradigms are embedded in a very similar way based on Scala's pattern matching constructs. Thus, parallels between the paradigms that were not previously obvious are revealed and the new concepts that must be learned by a programmer are limited.

A full comparison of Kiama with related work is beyond the scope of this paper. Nevertheless, it is important to note that the library has been heavily influenced by existing notations and implementations of both attribute grammars and rewriting. The attribute grammar facilities are modelled on those of the JastAdd system [4]. Kiama's term rewriting library is based on the Stratego language and library [5]. Kiama also shares some characteristics with other embeddings of these paradigms, most notably strategic programming in functional languages in the Strafunski [6] and Scrap Your Boilerplate projects [7].

Kiama is released under the GNU Lesser General Public License. Further information including binary distributions, code, documentation, examples and mailing lists can be found on the project site `http://kiama.googlecode.com`.

Outline

This paper presents an overview of Kiama's current capabilities with a focus on how the main features of the JastAdd and Stratego languages are supported via a lightweight embedding. (More detailed discussion of Kiama's relationship to JastAdd can be found in Sloane et al. [8]).

We proceed by developing implementations of typical processing tasks for a simply-typed version of the lambda calculus. This source language was chosen to be familiar and relatively simple, yet to provide processing tasks that are also relevant to other more complex languages.

Section 2 describes the version of lambda calculus used in the rest of the paper and how programs are represented as Scala data structures. Section 3

shows how Kiama's attribute grammar facilities can be used to define static program analyses. Section 4 implements various forms of evaluation mechanism as rewriting strategies. The paper concludes with a short discussion of Kiama's capabilities and future plans.

The paper presents the main code fragments necessary to achieve the desired effects. No knowledge of Scala is assumed, but experience with object-oriented programming and pattern-matching as in functional languages will be useful. We omit uninteresting scaffolding code that is necessary to turn these fragments into compilable Scala code. The complete source code can be found in the lambda2 example in the Kiama distribution.

2 A Typed Lambda Calculus

To keep things simple but realistic, we use a simply typed lambda calculus as the source language for the processing described in this paper. Figure 1 summarises the abstract syntax of the language.

The Scala version of the abstract syntax is a straight-forward encoding of the abstract syntax using Scala *case classes* (Figure 2). For most purposes, case classes operate as regular classes but also provide special construction syntax and pattern matching support similar to that provided for algebraic data types in functional languages. *Case objects* are the sole instances of anonymous singleton case classes.

As an example of construction, a tree fragment representing the lambda calculus expression

$$(\lambda x : Int\ .\ (\lambda y : Int\ .\ x + y - 2))\ 3\ 4$$

can be constructed in Scala by the expression

```
App (App (Lam ("x", IntType,
                Lam ("y", IntType,
                     Opn (SubOp,
                          Opn (AddOp, Var ("x"),
                                      Var ("y"))),
                          Num (2)))
           Num (3)),
      Num (4))
```

In later sections, we will pattern match against "constructors" such as `Lam` and `App` to deconstruct such expressions.

3 Attribution

Attribute grammars have been widely studied as a specification technique for describing computations on trees [9, 10]. In an attribute grammar, the context-free grammar of a language is augmented with *attribute equations* which define the values of attributes of tree nodes. In their purest form, attribute grammars

Expressions (e)	n	number
	v	variable
	$\lambda v : t \;.\; e$	lambda abstraction
	$e_1\ e_2$	application
	$e_1\ o\ e_2$	primitive operation
Types (t)	`int`	primitive integer type
	$t_1 \rightarrow t_2$	function type
Operations (o)	$+$	addition
	$-$	subtraction

Fig. 1. Abstract syntax of typed lambda calculus.

```
abstract class Exp
case class Num (n : Int) extends Exp
case class Var (i : Idn) extends Exp
case class Lam (n : Idn, t : Type, e : Exp) extends Exp
case class App (e1 : Exp, e2 : Exp) extends Exp
case class Opn (o : Op, e1 : Exp, e2 : Exp) extends Exp

type Idn = String

abstract class Type
case object IntType extends Type
case class FunType (t1 : Type, t2 : Type) extends Type

abstract class Op
case object AddOp extends Op
case object SubOp extends Op
```

Fig. 2. Scala data type to represent the lambda calculus abstract syntax.

have no notion of tree updates, so they are best suited to analysis of fixed structures and attributes can be understood as static properties.

Attribute grammar evaluation approaches can be divided into two broad categories: those that statically analyse attribute dependencies and those that wait until run-time. Kiama's approach is in the latter category [8]. It uses an evaluation mechanism similar to that apparently first used by Jourdan [11], a variant of which is also used in the JastAdd system [4]. Attributes are computed by functions that dynamically demand the values of any other necessary attributes. Attribute values are cached so that they do not needed to be re-evaluated if they are demanded again.

3.1 Free variables

As a simple example of using attribution to compute a useful property of a tree, consider *free variable analysis* of lambda calculus expressions. We want to calculate a set of the variables that are not bound in a supplied expression. A typical case-based definition of this analysis is as follows [12].

$$\begin{aligned} fv(n) &= \{\} \\ fv(v) &= \{v\} \\ fv(\lambda v : t\ .\ e) &= fv(e) - v \\ fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\ fv(e_1\ o\ e_2) &= fv(e_1) \cup fv(e_2) \end{aligned}$$

A Kiama version of this analysis uses standard Scala pattern matching to specify the same cases and build a Scala `Set` value (Figure 3). In attribute grammar terminology, `fv` is a *synthesised attribute* because it is defined in terms of attributes of the expression and its children.

In Figure 3, `==>` is a Kiama infix type constructor alias for Scala's generic partial function type. Thus, a function of type `T ==> U` transforms values of type `T` into values of type `U`, but may not be defined at all values of type `T`.

```
val fv : Exp ==> Set[Idn] =
  attr {
    case Num (_)          => Set ()
    case Var (v)          => Set (v)
    case Lam (v, _, e)    => fv (e) -- Set (v)
    case App (e1, e2)     => fv (e1) ++ fv (e2)
    case Opn (_, e1, e2) => fv (e1) ++ fv (e2)
  }
```

Fig. 3. Free variable attribute definition. `Exp ==> Set[Idn]` is the type of a partial function from expressions to sets of variable identifiers. The operators `--` and `++` are set difference and union, respectively. An underscore is a wildcard pattern which matches anything.

The code between braces in Figure 3 is standard Scala syntax for an anonymous pattern-matching partial function. Kiama's `attr` function wraps the pattern matching with the dynamic behaviour of non-circular attributes.

```
def attr[T,U] (f : T ==> U) : T ==> U
```

`attr (f)`, where `f` is a partial function between some types `T` and `U`, behaves just like `f` except that it caches its argument-result pairs and detects when a cycle is entered (i.e., when `f (t)` is requested while evaluating `f (t)`, for some node `t`).

The free variables of an expression `e` can now be referenced via a normal function application `fv (e)` or using Kiama's attribute access operator `->` as `e->fv`. The latter is designed to mimic traditional attribute grammar notations.

3.2 Name and type analysis

Free variable analysis is a very simple computation defined by a bottom-up traversal of the expression tree. *Name and type analyses* are examples of more complex processing that must be performed by compilers and many other source code analysis tools. An analysis of names is typically needed before type analysis can be performed. Our aim in this section is to analyse expressions such as $\lambda x : Int\ .\ (\lambda y : Int \rightarrow Int\ .\ y\ x)$ and determine that the application $y\ x$ is legal because y is a function from integer to integer and x is an integer.

It is not immediately obvious how to achieve the appropriate traversals of an expression tree to perform name and type analysis. Do we first perform a traversal for name analysis and then one for type analysis, or can we mix them somehow? The attribute grammar paradigm helps considerably with avoiding these questions because it enables us to concentrate on the dependencies between attributes. Dynamic scheduling of attribute computations will take care of the traversal. Therefore, we do not need to explicitly separate name and type analysis.

An environment-based analysis. First, we present a name and type analysis that uses explicit environment structures to keep track of the bound names and their types. An alternative where the tree itself holds this information is presented in the next section.

The `env` attribute computes an environment for a given expression consisting of all variables that are visible at that expression and their types. The environment is represented by a list, with the interpretation that earlier entries hide later ones, thereby implementing variable shadowing.

In contrast to the `fv` attribute which was defined by matching on the node itself, the `env` attribute is defined by cases on its parent. In other words, the names that are visible at a node depend on the node's context. We have three cases: a) at the top of the tree (null parent) nothing is visible, b) inside a lambda expression, the visible names are whatever is visible at the lambda node plus the name bound at that node, and c) in all other cases, the names visible at a

node are just those that are visible at the node's parent. These cases are easily specified by pattern matching (Figure 4). In attribute grammar terms, `env` is an *inherited attribute.*[1]

```
val env : Exp ==> List[(Idn,Type)] =
  attr {
    case e =>
      (e.parent) match {
        case null               => List ()
        case p @ Lam (x, t, _) => (x,t) :: p->env
        case p : Exp            => p->env
      }
  }
```

Fig. 4. Definition of environment attribute as list of bound variables and their types. Scala's `match` construct performs pattern matching. A pattern `p @ patt` matches against the pattern `patt` and, if successful, binds `p` to the matched value. A pattern of the form `p : T` succeeds if the value being matched is of type `T`, in which case it binds `p` to the value. An underscore is a pattern that matches anything. `::` is the List prepend operation.

Kiama provides fields called *structural properties* that give generic access to the tree structure. For example, the `parent` field of `e` used in Figure 4 is a structural property that provides access to the parent of any node, or null if the node is the root. The other structural properties provided by Kiama are `isRoot` for all nodes, and `prev`, `next`, `isFirst` and `isLast` for nodes occurring in sequences. The structural properties are provided automatically to any class that inherits Kiama's `Attributable` trait, as in

```
abstract class Exp extends Attributable
```

With `env` in hand, we can define the `tipe` attribute[2] that gives the type of any expression (Figure 5). Unlike traditional typing rules, the environment is not passed, because it can be accessed directly using the `env` attribute as needed. There are five cases:

a) a number has integer type,
b) a lambda expression $\lambda x : t \;.\; e$ has type $t \to t_e$ where t_e is the type of e,
c) an application of a function of type $t_1 \to t_2$ to an expression of type t_1 is of type t_2,
d) the operands and result of an operation are integers, and
e) the type of a variable is the type associated with that variable name in the environment.

[1] In some cases, not shown in this overview, it is useful to match on both the node and its parent. Therefore, the synthesised versus inherited distinction is not particularly meaningful in Kiama, since each attribute definition is free to access any part of the tree that it needs.

[2] `type` cannot be used since it is a Scala keyword.

If none of these cases apply, a typing error is reported using Kiama's message facility and an error type of `IntType` is returned. (Of course, this approach may lead to spurious errors. A more robust implementation would return a dedicated error type and ensure that values of the error type were acceptable in any context.)

```
val tipe : Exp ==> Type =
  attr {
    case Num (_) =>
      IntType

    case Lam (_, t, e) =>
      FunType (t, e->tipe)

    case App (e1, e2) =>
      e1->tipe match {
        case FunType (t1, t2) if t1 == e2->tipe =>
          t2
        case FunType (t1, t2) =>
          message (e2, "need " + t1 + ", got " +
                       (e2->tipe))
          IntType
        case _ =>
          message (e1, "application of non-function")
          IntType
      }

    case Opn (op, e1, e2) =>
      if (e1->tipe != IntType)
        message (e1, "need Int, got " + (e1->tipe))
      if (e2->tipe != IntType)
        message (e2, "need Int, got " + (e2->tipe))
      IntType

    case e @ Var (x) =>
      (e->env).find { case (y,_) => x == y } match {
        case Some ((_, t)) => t
        case None =>
          message (e, "'" + x + "' unknown")
          IntType
      }
  }
```

Fig. 5. Definition of the expression type attribute. Kiama's `message` operation records a message associated with a particular tree node. Scala's `find` method searches a list using the predicate provided as an argument and returns an `Option[T]` value, where `T` is the list element type. A value of type `Option[T]` is either `Some (t)` for some value `t` of type `T`, or it is `None`.

A reference-based analysis. In the environment-based analysis, we reuse the type nodes of the tree when constructing the environment (in the `Lam` case).

We can go further and do away with the environment completely by observing that each binding can be represented by the lambda expression in which it is created. This kind of observation is at the heart of Hedin's Reference Attribute Grammars [13], which can be achieved in Kiama with the facilities we have seen already.

All of the cases for the `tipe` attribute stay the same as in the environment version, except for the variable case (Figure 6). Instead of looking up the name in the environment, we define a `lookup` attribute that traverses the tree to find the name if it can. There are three cases: a) we are examining a lambda expression that defines the name we are looking for, so return that lambda expression, b) we are at the root of the tree, so report that we didn't find a binder for the name, and c) ask the parent to lookup the name. `lookup` is therefore a *parameterised, reference attribute* in attribute grammar terminology.[3] In the variable case of `tipe` we can now use `lookup` to find the binder of the name, if there is one.

In `lookup`, the parent reference `e.parent[Exp]` requires the type annotation `Exp` because `parent` is generic and the compiler is not able to infer that expressions only ever occur as children of expressions. The necessity for this annotation reveals a limitation in the embedding approach due to full grammar knowledge not being available when the abstract syntax is implemented as a class hierarchy. The type annotation results in a cast to the given type and it is up to the developer to ensure that the cast cannot fail. More discussion of this issue can be found in Sloane et al. [8].

4 Rewriting

Kiama's rewriting library is closely modelled on the Stratego rewriting language [5]. Stratego uses a general notion of a *rewriting strategy* that takes as input a term representing a tree structure, and either succeeds, producing a (possibly) rewritten term, or fails. Stratego has a rich language of strategy combinators and library strategies that achieve choice, iteration and other more complex term traversal patterns.

The aim for this part of Kiama was to see how much of Stratego could be realised using a pure embedding approach, in contrast to the standard implementation which compiles to C. As this section shows, most of Stratego can be easily encoded. Our encoding is based around a functional abstraction similar to that used for attribute equations in the previous section. Standard Scala pattern matching can be used within rewrite rules. Implementations of the Stratego combinators and library strategies enable most Stratego programs to be written using almost the same syntax.

This section presents examples of using Kiama's rewriting library to evaluate lambda calculus expressions, based on Stratego versions of the same [14].

[3] This use of a parameterised attribute defined by a Scala function is simple, but it may not provide the desired caching behaviour, since the parameter value is not included in the cache key. Kiama also provides a variant of `attr` that can be used if more advanced caching is important.

```
def lookup (name : Idn) : Exp ==> Option[Lam] =
  attr {
    case e @ Lam (x, t, _) if x == name =>
      Some (e)
    case e if e.isRoot =>
      None
    case e =>
      e.parent[Exp]->lookup (name)
  }

val tipe : Exp ==> Type =
  attr {

    ...

    case e @ Var (x) =>
      (e->lookup (x)) match {
        case Some (Lam (_, t, _)) =>
          t
        case None =>
          message (e, "'" + x + "' unknown")
          IntType
      }

  }
```

Fig. 6. Definition of the name lookup attribute and the new case for name and type analysis of variables.

4.1 Evaluation

The evaluation strategies fit into a general framework. The interface to an evaluator is a function `eval` that takes an expression and returns the expression that is the result of evaluation.

```
def eval (exp : Exp) : Exp =
  rewrite (s) (exp)

val s : Strategy
```

Evaluation is achieved by rewriting with the strategy `s` which is defined in various ways in the following sections. `rewrite` applies its strategy argument to its term argument. If the strategy succeeds, `rewrite` returns the resulting term, otherwise, it returns the original argument.

4.2 Basic reduction

The basic evaluation rule for lambda calculus is *beta reduction* [12].

$$(\lambda x : t \;.\; e_1)\; e_2 \rightarrow [e_2/x]e_1$$

where $[e_2/x]e_1$ means capture-free substitution of e_2 for occurrences of the variable x in e_1. Primitive operations can be evaluated by reduction rules that use operations in the meta-language.

Figure 7 shows an encoding of these rules as strategies in Kiama. Each rule is written as a pattern matching function on the relevant tree structure.[4] The pattern match is wrapped by a call to Kiama's `rule` function that converts the function into a strategy.

```
def rule (f : Term ==> Term) : Strategy
```

A `Strategy` is a function from `Term` to `Option[Term]`. The `Option` wrapper is used to represent success and failure. `rule` lifts a partial function `f` to the `Strategy` type, mapping undefinedness of `f` to `None`, representing failure of the strategy. In other words, `rule (f)`, for some partial function `f`, when applied to a term `t`, succeeds with the result of `f (t)`, if `f` is defined at `t`, otherwise it fails.

```
val s =
  reduce (beta + arithop)

val beta =
  rule {
    case App (Lam (x, _, e1), e2) =>
      substitute (x, e2, e1)
  }

val arithop =
  rule {
    case Opn (op, Num (l), Num (r)) =>
      Num (op.eval (l, r))
  }
```

Fig. 7. Definition of simple reduction strategies. The function `substitute` is assumed to implement capture-free substitution. Each primitive operator is assumed to have a method `eval` that evaluates that operator on two integers and returns the result.

The `beta` and `arithop` strategies are combined in Figure 7 to form `s` using the non-deterministic choice operator `+`. Finally, the library strategy `reduce` is used to repeatedly apply the basic strategies to the subject term until a fixed point is reached.

4.3 The `reduce` strategy

The definition of `reduce` shows both the power of the Stratego language for combining strategies, but also the relatively clean way that this language can be embedded into Scala. In Stratego, `reduce` is defined in terms of other library strategies and basic combinators as

```
try (s)    = s <+ id
```

[4] While the different rules could be combined into a single one, we prefer to keep them separate to enable more flexible reuse.

```
repeat (s) = try (s; repeat (s))
reduce (s) = repeat (rec x (some (x) + s))
```

where the new Stratego constructs are

- the identity strategy (`id`) which always succeeds without changing the subject term,
- deterministic choice (`<+`) where the second strategy is only applied to the subject term if the first strategy fails,
- sequential composition (`;`), where the second strategy is applied to the result of a successful invocation of the first,
- definition of a locally recursive binding of `x` by `rec x`, and
- the primitive traversal combinator `some` whose result succeeds if the argument strategy succeeds on at least one child of the subject term.[5]

Thus, we can see that `try` attempts to apply its argument strategy but leaves the term unchanged if that strategy fails. `repeat` applies a strategy repeatedly until it fails. `reduce` repeatedly applies a strategy to sub-terms and the subject term itself until all of those applications fail, upon which it succeeds with the most recent result.

Stratego programs are built up in this way from a collection of primitives and a large library. The result is an extremely expressive language of tree traversal and transformation. Similar power can be achieved in Kiama using notations that are very similar to those of Stratego, even though we rely entirely on Scala syntax and concepts.

Figure 8 shows the Kiama version of the library strategies needed for the basic reduction example. Scala's ability to define methods with symbolic names means that the primitive combinators `<+` and `+` can be provided as `Strategy` methods; we use `<*` for sequencing, since semicolon is already claimed for other purposes by the Scala syntax. Similarly, `try` is renamed `attempt` since the former is a Scala keyword. Other than these cosmetic changes, the main differences between the two versions are the inclusion of the type information and a more verbose definition for the recursive value `x` in `reduce`.

4.4 Explicit substitution

Instead of relying on a separate `substitute` function to implement the core of the beta reduction rule, explicit substitutions can be used to bring the entire evaluation process into the rewriting paradigm.

Figure 9 shows how an explicit substitution version can be written in Kiama. First, a new `Let` tree construct is declared to represent substitutions. The substitution $[e_2/x]e_1$, where x has type t, will be represented by the expression `Let ("x", t, e2, e1)`

[5] Stratego and Kiama also have `all` and `one` that require success on all children or one child, respectively.

```
def attempt (s : => Strategy) : Strategy =
  s <+ id

def repeat (s : => Strategy) : Strategy =
  attempt (s <* repeat (s))

def reduce (s : => Strategy) : Strategy = {
  def x : Strategy = some (x) + s
  repeat (x)
}
```

Fig. 8. Kiama version of Stratego library combinators. A parameter type preceded by `=>` indicates a pass-by-name mode.

`s` is now defined in terms of a `lambda` strategy which in turn combines beta reduction (modified to produce an explicit substitution), primitive evaluation (unchanged) and a set of new strategies that implement substitution. `subsVar` actually performs substitution on a variable reference, whereas the others propagate substitutions inward. (As before, a single rule could be used instead of these reusable pieces.)

4.5 Innermost evaluation

Using `reduce` has an efficiency penalty because it repeats its search for a reducible expression starting from the top of the whole expression each time. An *innermost evaluation* reduces sub-terms before trying to reduce the subject term. Stratego's `innermost` library strategy is defined as follows in terms of a more general `bottomup` traversal strategy.

```
innermost (s) = bottomup (try (s; innermost (s)))
bottomup (s)  = all (bottomup (s)); s
```

both of which are defined in an analogous way in the Kiama library.

`innermost` can be used with the `lambda` strategy defined in the previous section to achieve a more efficient evaluation. In Kiama syntax, we have

```
val s = innermost (lambda)
```

4.6 Eager evaluation

An innermost evaluation is still not very realistic since in a programming language implementation based on lambda calculus it is unlikely that reductions will be performed inside the body of a lambda expression until that expression is applied to an argument. *Eager evaluation* reduces the arguments of applications before the reduction of applications.

An evaluation strategy to express this pattern of evaluation is as follows.

```
val s : Strategy =
  attempt (traverse) <* attempt (lambda <* s)
```

```
case class Let (name : Idn, tipe : Type, exp : Exp,
                body : Exp) extends Exp

val s =
  reduce (lambda)

val lambda =
  beta + arithop + subsNum + subsVar + subsApp +
  subsLam + subsOpn

val beta =
  rule {
    case App (Lam (x, t, e1), e2) =>
      Let (x, t, e2, e1)
  }

val subsNum =
  rule {
    case Let (_, _, _, e : Num) => e
  }

val subsVar =
  rule {
    case Let (x, _, e, Var (y)) if x == y => e
    case Let (_, _, _, v : Var)           => v
  }

val subsApp =
  rule {
    case Let (x, t, e, App (e1, e2)) =>
      App (Let (x, t, e, e1), Let (x, t, e, e2))
  }

val subsLam =
  rule {
    case Let (x, t1, e1, Lam (y, t2, e2)) if x == y =>
       Lam (y, t2, e2)
    case Let (x, t1, e1, Lam (y, t2, e2)) =>
      val z = freshvar ()
      Lam (z, t2, Let (x, t1, e1,
                       Let (y, t2, Var (z),
                            e2)))
  }

val subsOpn =
  rule {
    case Let (x, t, e1, Opn (op, e2, e3)) =>
      Opn (op, Let (x, t, e1, e2), Let (x, t, e1, e3))
  }
```

Fig. 9. Definition of reduction with explicit substitutions. In `subsLam`, `freshvar` is a helper function that returns a unique variable name each time it is called.

First, we traverse the expression to evaluate any parts of it that should be evaluated before reduction at the top-level of the expression is attempted. The `lambda` strategy from earlier can be reused and augmented with a simple traversal strategy that controls exactly which sub-terms are reduced first.

```
val traverse : Strategy =
  rule {
    case App (e1, e2) =>
      App (eval (e1), eval (e2))
    case Let (x, t, e1, e2) =>
      Let (x, t, eval (e1), eval (e2))
    case Opn (op, e1, e2) =>
      Opn (op, eval (e1), eval (e2))
  }
```

In this version of `traverse` we evaluate eagerly, so that both sides of applications, the bound expressions and bodies of substitutions and the operands of primitives are evaluated. Forms that are not to be traversed do not need to be mentioned.

4.7 Congruences

Stratego provides a short-hand *congruence* notation for expressing traversal strategies of this kind. For example, if C is a node constructor with two arguments and s_1 and s_2 are strategies, then $C(s_1, s_2)$ is a congruence for C. It matches any C node, applies s_1 to the first component of the node, applies s_2 to the second component, and, if both s_1 and s_2 succeed, creates a new C node containing their results in the first and second components, respectively. If either s_1 or s_2 fail, then $C(s_1, s_2)$ fails.

`traverse` from the previous section can be written using Stratego congruences as follows.

```
App (s, s) + Let (id, id, s, s) + Opn (id, s, s)
```

The effect is to recursively evaluate those parts of the structure where `s` appears and leave untouched those parts where `id` appears.

Automatic support for congruences appears to beyond a pure embedding approach, since it requires knowledge of the abstract syntax. As a partial measure, Kiama helps developers of abstract syntaxes write their own congruences.[6] For example, a congruence for `App` can be written in Kiama as follows.

```
def App (s1 : => Strategy,
         s2 : => Strategy) : Strategy =
  rulefs {
    case _ : App => congruence (s1, s2)
  }
```

This definition overloads `App` to take strategy arguments. The pattern match restricts attention to `App` nodes and a Kiama library function `congruence` returns a strategy that implements the semantics of the congruence using `s1` and `s2`. `rulefs` is a variant of `rule` that takes a function returning a strategy instead of

[6] We plan to generate congruences from a description of the abstract syntax in a future version of Kiama.

the usual function that returns a term. With this congruence definition, eager evaluation can be defined simply, without the supplementary `traverse` strategy.

```
val s : Strategy =
  attempt (App (s, s) + Let (id, id, s, s) +
           Opn (id, s, s)) <*
  attempt (lambda <* s)
```

4.8 Lazy evaluation

Finally, we consider lazy evaluation where as much as possible is left un-reduced until a beta reduction is performed. Only the traversal strategy needs to change; the new traversal refrains from evaluating application arguments and let-bound expressions too early. (A full lazy evaluation method would also include sharing of computed values, which we omit here.) The change is restricted to the congruences, where `id` now appears in the positions for arguments to applications and bound expressions in `Let` constructs.

```
val s : Strategy =
  attempt (App (s, id) + Let (id, id, id, s) +
           Opn (id, s, s)) <*
  attempt (lambda <* s)
```

5 Conclusion

The examples in this paper are typical language processing problems: static analysis and evaluation by transformation. We have seen that these problems can be solved easily using embeddings of attribute grammars and strategic-based rewriting in a general purpose language. The attribute grammar embedding achieves a substantial proportion of the functionality of JastAdd and the rewriting embedding is very faithful to the Stratego language design. These embeddings can also be employed to solve other language processing tasks such as desugaring, interpretation, code generation, and optimisation.

The similarities between the embeddings of attribute grammars and rewriting show that these two paradigms are alike in ways that have not been appreciated to date. In each case, a simple functional interface, provided by `attr` and `rule`, suffices to hide the complexities of the representation of attributes and strategies. Apart from calling these functions, a Kiama programmer uses standard Scala constructs to define attribute equations and rewriting rules. Thus, Kiama's version of these paradigms is particularly lightweight compared to standalone generator-based systems such as JastAdd and Stratego. This lightweight nature makes it more accessible to mainstream developers who would otherwise not be exposed to these high-level processing paradigms. Moreover, since Kiama is pure Scala, it automatically gains advantage from existing Scala tools such as IDE support for editing and debugging, further simplifying the adoption process.

Kiama has some advanced capabilities that have not been presented here. The attributes used in this paper cannot have cyclic dependencies (i.e., depend on

themselves). In some situations such cyclic dependencies are useful, particularly in analysis problems where a solution is found by computing until a fixed point is reached. See Sloane et al. [8] for an example that computes variable liveness information using a variant of `attr` designed to handle cyclic dependencies. Section 3.2 used a higher-order attribute which was a reference to an existing tree node; Kiama also allows higher-order attributes that refer to new nodes and supports *forwarding* [15] to redirect attribute evaluations automatically to higher-order attributes. Finally, Kiama includes *attribute decorators* [16] that can express patterns of attribute propagation.

Work on Kiama continues. Of particular interest is the interaction between the two paradigms, such as using the free variables attribute during rewriting. This kind of combination raises questions about the validity of attribute values after a rewriting step. We are exploring methods for removing the necessity for type casting of the generic structural properties such as `parent` due to a lack of knowledge about the tree structure. We are also investigating features such as collection attributes [17, 18] and better support for attribute modularity. Following Stratego, Kiama's strategies are currently largely untyped, except that Scala's type rules prevent ill-typed terms from being created. Typed strategies will come to Kiama soon. It would also be useful to have some way of specifying pattern matching on objects using concrete syntax [19, 20].

Acknowledgements

The author thanks the Software Engineering Research Group at the Technical University of Delft for hosting the study leave during which the Kiama project was initiated. That visit was supported by the Dutch NWO grant 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*. Lennart Kats and Eelco Visser co-developed the attribute grammar facilities described here and provided the author with instruction in the arts of Stratego. Charles Consel, Mark van den Brand, members of IFIP WG 2.11 and anonymous reviewers have also provided useful feedback on the Kiama project.

References

[1] Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press (2008)
[2] Odersky, M.: Scala language specification, Version 2.7. Programming Methods Laboratory, EPFL, Switzerland. (2009)
[3] Kastens, U., Sloane, A.M., Waite, W.M.: Generating Software from Specifications. Jones and Bartlett, Sudbury, MA (2007)
[4] Hedin, G., Magnusson, E.: Jastadd: an aspect-oriented compiler construction system. Sci. Comput. Program. **47** (2003) 37–58
[5] Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C., et al., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. Springer-Verlag (2004) 216–238
[6] Lammel, R., Visser, J.: A Strafunski Application Letter. Proc. of Practical Aspects of Declarative Programming (PADL'03) **2562** (2003) 357–375

[7] Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices **38** (2003) 26–37 Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
[8] Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In Vinju, J., Ekman, T., eds.: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (to appear in ENTCS). (2010)
[9] Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. Volume 323 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1988)
[10] Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. **27** (1995) 196–255
[11] Jourdan, M.: An optimal-time recursive evaluator for attribute grammars. In: Proceedings of the International Symposium on Programming, Springer (1984) 167–178
[12] Reynolds, J.C.: Theories of Programming Languages. Camridge University Press (1998)
[13] Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) **24** (2000) 301–317
[14] Dolstra, E., Visser, E.: Building interpreters with rewriting strategies. In: Proceedings of the 2nd Workshop on Language Descriptions, Tools and Applications. Volume 65 of Electronic Notes in Theoretical Computer Science. (2002) 57–76
[15] Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In Horspool, R.N., ed.: Proceedings of the 11th International Conference on Compiler Construction. Volume 2304 of Lecture Notes in Computer Science., Springer-Verlag (2002) 128–142
[16] Kats, L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: Proceedings of the International Conference on Compiler Construction. Number 5501 in Lecture Notes in Computer Science, Springer-Verlag (2009) 142–157
[17] Boyland, J.T.: Descriptional Composition of Compiler Components. PhD thesis, University of California, Berkeley (1996) Available as technical report UCB//CSD-96-916.
[18] Magnusson, E., Ekman, T., Hedin, G.: Extending attribute grammars with collection attributes - evaluation and applications. In: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE Press (2007)
[19] van den Brand, M.G.J., van Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a component-based language development environment. In Wilhelm, R., ed.: Proceedings of International Conference on Compiler Construction. Volume 2027 of Lecture Notes in Computer Science., Springer-Verlag (2001) 365–370
[20] Bravenboer, M., Visser, E.: Concrete syntax for objects. In Schmidt, D.C., ed.: Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04), Vancouver, Canada, ACM Press (2004) 365–383