A Pure Object-Oriented Embedding of Attribute Grammars

Anthony M. Sloane
Macquarie University

Lennart Kats   Eelco Visser
Delft University of Technology

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.

The Eli System

A black-box language processor generation system that integrates off-the-shelf tools. Under development since the late 1980s by groups at Colorado, Paderborn and Macquarie.

- Many different DSLs as specification notations
- Lots of generators, some custom-built and some off-the-shelf
- A complex integration problem, both at the specification level but also at the generator level
- A large library of reusable code and specifications
- High-level execution monitoring in terms of domain model

Further information about Eli

Generating Software from Specifications, Uwe Kastens, Anthony M. Sloane, and William M. Waite, Jones and Bartlett, 2007

which includes

- short and long case studies,
- coverage of "peripheral" topics: specification structure, manufacturing and execution monitoring

Download:

http://sourceforge.net/projects/eli-project

What Now?

Report of a workshop on Future Directions of Programming Languages [1995]:

> *Our view is that many special-purpose languages are far more special than their purpose requires. They are often designed with just the right primitives and "first-order" syntax, but with an overarching language structure that is feeble and ad hoc.*

as reported in [Kamin96]

Explore trade-offs between specialised design of a new language and the power of a general-purpose language

Use specialised notations only where general purpose notations are not appropriate

The Kiama Library

An experiment in embedding language processing formalisms in the Scala programming language.

Currently includes:

- packrat parsing combinators
- strategy-based term rewriting
- dynamically-scheduled attribute grammars

First public release coming soon.

http://plrg.science.mq.edu.au/projects/show/kiama

The Kiama Library

An experiment in embedding language processing paradigms in the Scala programming language.

Currently includes:

- packrat parsing combinators
- strategy-based term rewriting
- dynamically-scheduled attribute grammars

First public release coming soon.

http://plrg.science.mq.edu.au/projects/show/kiama

Scala Programming Language

Odersky et al, Programming Methods Laboratory, EPFL, Switzerland

Main characteristics:

- object-oriented at core with functional features
- statically typed, local type inference
- scalable: scripting to large system development
- runs on JVM, interoperable with Java

Talk Outline

Review of attribute grammars and their implementation.

Examples of typical attribute grammars written using Kiama:

- repmin
- variable liveness.

An overview of the Kiama attribute grammar implementation.

Discussion, including:

- comparison of a Kiama attribute grammar with a JastAdd equivalent.

Attribute Grammars

Attributes are properties of tree nodes.

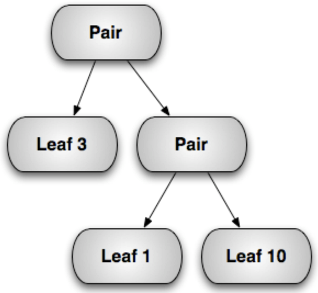
Attribute equations associated with context-free grammar productions describe how attribute values are related to other attribute values.

A declarative formalism from which evaluation strategies can be automatically determined.

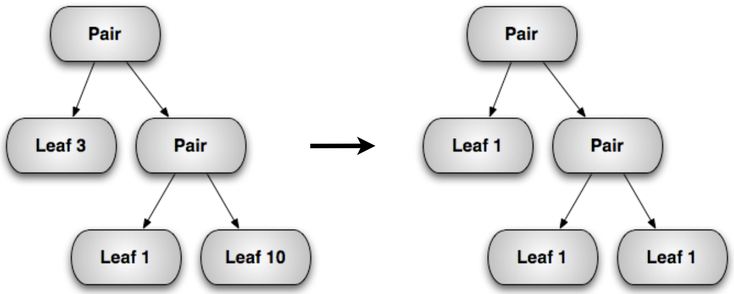
Static attribute scheduling: determine at generation time a tree traversal strategy that will enable the attributes to be evaluated in an appropriate order.

Dynamic attribute scheduling: evaluate only those attributes that are needed to compute a property of interest.

Repmin



Repmin



Repmin : tree structure

```
abstract class Tree extends Attributable

case class Pair (left : Tree, right : Tree) extends Tree

case class Leaf (value :  Int) extends Tree

val t = Pair (Leaf (3), Pair (Leaf (1), Leaf (10)))
```

Repmin : local and global minima

```
val locmin : Tree ==> Int =
  attr {
    case Pair (l, r) => (l->locmin) min (r->locmin)
    case Leaf (v)  => v
  }

val globmin : Tree ==> Int =
  attr {
    case t if t isRoot => t->locmin
    case t             => t.parent[Tree]->globmin
  }
```


Repmin : result tree

```
val repmin : Tree ==> Tree =
  attr {
    case Pair (l, r)  => Pair (l->repmin, r->repmin)
    case t : Leaf     => Leaf (t->globmin)
  }
```


Variable Liveness

	In	Out
y = v;	{v, w}	{v, w, y}
z = y;	{v, w, y}	{v, w}
x = v;	{v, w}	{v, w, x}
while (x) {	{v, w, x}	{v, w, x}
x = w;	{v, w}	{v, w}
x = v;	{v, w}	{v, w, x}
}		
return x;	{x}	

Liveness : tree structure

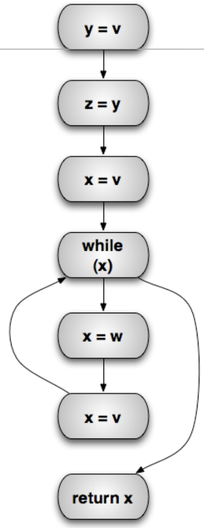
```
type Var = String

abstract class Stm extends Attributable

case class Assign (left : Var, right : Var) extends Stm
case class While (cond : Var, body : Stm) extends Stm
case class If (cond : Var, tru : Stm, fls : Stm) extends Stm
case class Block (stms : Stm*) extends Stm
case class Return (ret : Var) extends Stm
case class Empty () extends Stm
```

Liveness : control flow graph

```
y = v;
z = y;
x = v;
while (x) {
   x = w;
   x = v;
}
return x;
```



Liveness : successor nodes

```
val succ :  Stm ==> Set[Stm] =
  attr {
    case If (_, s1, s2)       => Set (s1, s2)
    case t @ While (_, s)  => t->following + s
    case Return (_)          => Set ()
    case Block (s, _*)       => Set (s)
    case s                       => s->following
  }
```


Liveness : following nodes

```
val following :  Stm ==> Set[Stm] =
  childAttr {
    case s => {
      case t @ While (_, _)                 => Set (t)
      case b @ Block (_*) if s isLast => b->following
      case Block (_*)                         => Set (s.next)
      case _                                    => Set ()
    }
  }
```


Liveness : variable uses and definitions

```
val uses : Stm ==> Set[String] =
  attr {
    case If (v, _, _)       => Set (v)
    case While (v, _)    => Set (v)
    case Assign (_, v) => Set (v)
    case Return (v)      => Set (v)
    case _                   => Set ()
  }

val defines : Stm ==> Set[String] =
  attr {
    case Assign (v, _) => Set (v)
    case _                   => Set ()
  }
```

Liveness : in and out variables

$$in(s) = uses(s) \cup (out(s) \setminus defines(s))$$

$$out(s) = \bigcup_{x \in succ(s)} in(x)$$

Liveness : in and out variables

$$in(s) = uses(s) \cup (out(s) \setminus defines(s))$$

$$out(s) = \bigcup_{x \in succ(s)} in(x)$$

```
val in : Stm ==> Set[String] =
  circular (Set[String]()) {
    case s => uses (s) ++ (out (s) -- defines (s))
  }

val out : Stm ==> Set[String] =
  circular (Set[String]()) {
    case s => (s->succ) flatMap (in)
  }
```


Implementation

Attributes

- partial function objects from tree nodes to attribute values
- maintain an object-local cache mapping tree nodes to value

Attribute value notation

- sugar for a function call
- tree -> a is the same as a (tree)

Tree structure is visible to attributes via node properties

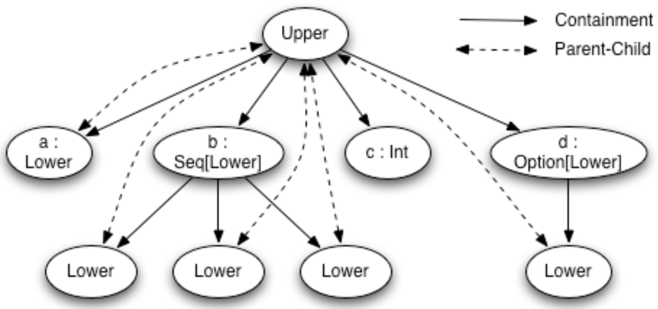
- an abstraction of the Scala tree structure

Tree structure

```
case class Upper (a : Lower, b : Lower*, c : Int,
                  d : Option[Lower])
case class Lower (...)
```

Tree structure

```
case class Upper (a : Lower, b : Lower*, c : Int,
                  d : Option[Lower])
case class Lower (...)
```



Cached attributes

```
def attr[T <: Attributable,U] (f : T ==> U) : T ==> U =
  new CachedAttribute (f)
```


Cached attributes

```
def attr[T <: Attributable,U] (f : T ==> U) : T ==> U =
  new CachedAttribute (f)

class CachedAttribute[T,U] (f : T ==> U) extends (T ==> U) {
  val memo = new IdentityHashMap[T,Option[U]]
  def apply (t : T) : U =
    memo.get (t) match {
      case None => memo (t) = None
                   val u = f (t)
                   memo (t) = Some (u)
                   u
      case Some (Some (u)) => u
      case Some (None)     => error ("Cycle detected")
    }
}
```


Other kinds of attribute

Child ("inherited") attributes

As for regular attributes but also pattern match on parent node.

Circular attributes

Use the basic circular evaluation algorithm from "Circular Reference Attributed Grammars - their Evaluation and Applications", by Magnusson and Hedin from LDTA 2003.

Parameterised attributes

Uncached and constant attributes

Discussion

Storage in tree nodes vs attribute-centred storage

Built-in laziness vs roll-your-own memoisation

Custom front-end vs general purpose language

Completeness checking vs sub-typing and sealed types

Context-free grammar vs GPL type system

Implicit composition vs explicit composition

Implementation size: around 230 lines of code

Preliminary Benchmark

Small experiment to validate expressibility and efficiency.

Encode an existing JastAdd picoJava specification

18 abstract grammar productions and 10 attributes
name analysis and inheritance cycle detection

Kiama implementation of same attribute equations has similar performance to JastAdd-generated implementation running on same JVM.

Conclusion

Kiama attribution library

lightweight, natural and easy to understand
competitive in expressiveness and reasonable performance

Scala

sweetspot combination of functional and object-oriented features, enables a simple implementation

Future work

attribute kinds: collections, forwarding, ...
modular attribute definitions