Lightweight Language Processing in Kiama

Anthony M. Sloane

Programming Languages Research Group
Department of Computing, Macquarie University
Sydney, Australia

Anthony.Sloane@mq.edu.au
http://www.comp.mq.edu.au/~asloane
http://plrg.science.mq.edu.au

Supported by The Netherlands 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.

MACQUARIE UNIVERSITY | FACULTY OF SCIENCE

The Kiama Library

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

Currently includes:

- packrat parsing combinators (soon to be removed)
- strategy-based term rewriting
- dynamically-scheduled attribute grammars

Project web site:

http://kiama.googlecode.com

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

Stratego

A powerful term rewriting language based on

- primitive match, build, sequence and choice operators
- rewrite rules built on the primitives
- generic traversal operators to control application rules
- an implementation by translation to C

Deployed for many program transformation problems including DSL implementation, compiler optimisation, refactoring and web application development.

http://strategoxt.org

Strategy

A transformation of a term that either

- succeeds producing a new term, or
- fails

```
abstract class Strategy extends (Term => Option[Term])

abstract class Option[A]
case class Some[A] (val a : A) extends Option[A]
case object None extends Option[Nothing]
```

Abstract Syntax

```
type Idn = String

abstract class Exp

case class Num (value : Int) extends Exp
case class Var (name : Idn) extends Exp
case class Lam (name : Idn, tipe : Type, body : Exp)
               extends Exp
case class App (l : Exp, r : Exp) extends Exp
case class Opn (op : Op, left : Exp, right : Exp)
               extends Exp
case class Let (name : Idn, tipe : Type, exp : Exp,
                body : Exp) extends Exp
```

Term Examples

```
// 1 + 3

val a = Opn(AddOp,Num(1),Num(3))

// \x : Int . x + y

val b = Lam("x",IntType,Opn(AddOp,Var("x"),Var("y")))

// (\x : Int -> Int . x 5) 7

val c = App(Lam("x",FunType(IntType,IntType),
                App(Var("x"),Num(5))),
            Num(7))
```

Applying Strategies

A strategy is just a function, so it can be applied directly to a term.

```
val s : Strategy
val t : Term
s (t)
```

rewrite can be used to ignore failure.

```
def rewrite (s : => Strategy) (t : Term) : Term

rewrite (s) (t)
```

Basic Strategies

Always succeed with no change.

```
val id : Strategy
```

Always fail.

```
val failure : Strategy
```

Succeed if the current term is equal to t.

```
def term (t : Term) : Strategy
```

Always succeed, changing the term to t.

```
implicit def termToStrategy (t : Term) : Strategy
```

Rewrite Rules

Rewrite rules are defined by Scala partial functions.

```
def rule (f : PartialFunction[Term,Term]) : Strategy
```

A rewrite rule to evaluate arithmetic operations using Scala's case syntax for partial functions.

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

Combining Strategies

Methods on the Strategy class allow strategies to be combined.

`p <* q`	sequence
`p <+ q`	deterministic choice
`p + q`	non-deterministic choice
`p < q + r`	guarded choice

Scala has a flexible naming convention for methods and allows the period to be omitted in a call.

`p <+ q <* r` is just `(p.<+(q)).<*(r)`

Library Strategies (1)

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

def not (s : => Strategy) : Strategy =
    s < failure + id

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

Library Strategies (2)

```
def topdown (s : => Strategy) : Strategy =
    s <* all (topdown (s))

def oncetd (s : => Strategy) : Strategy =
    s <+ one (oncetd (s))

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

Lambda Calculus with Meta-level Substitution

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

val evals = reduce (beta + arithop)

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

def substitute (x : Idn, e2: Exp, e1 : Exp) : Exp
```

Lambda Calculus with Explicit Substitution

```
val evals = reduce (lambda_es)

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

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

Explicit Substitution (1)

```
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
    }
```

Explicit Substitution (2)

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

Explicit Substitution (3)

```
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)))
    }
```

Attribute Grammars

Attributes are properties of tree nodes.

Attribute equations are associated with context-free grammar productions to 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 all attributes to be evaluated in an *appropriate* order.

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

Attribute Grammars in Kiama

Joint work with Lennart Kats and Eelco Visser (TU Delft)

Attribute

partial function from tree node to attribute value
maintains an attribute-local cache

Attribute value notation

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

Augmented tree structure is visible to attributes via node properties including parent and next, prev, isFirst, isLast for nodes in sequences

Variable Liveness

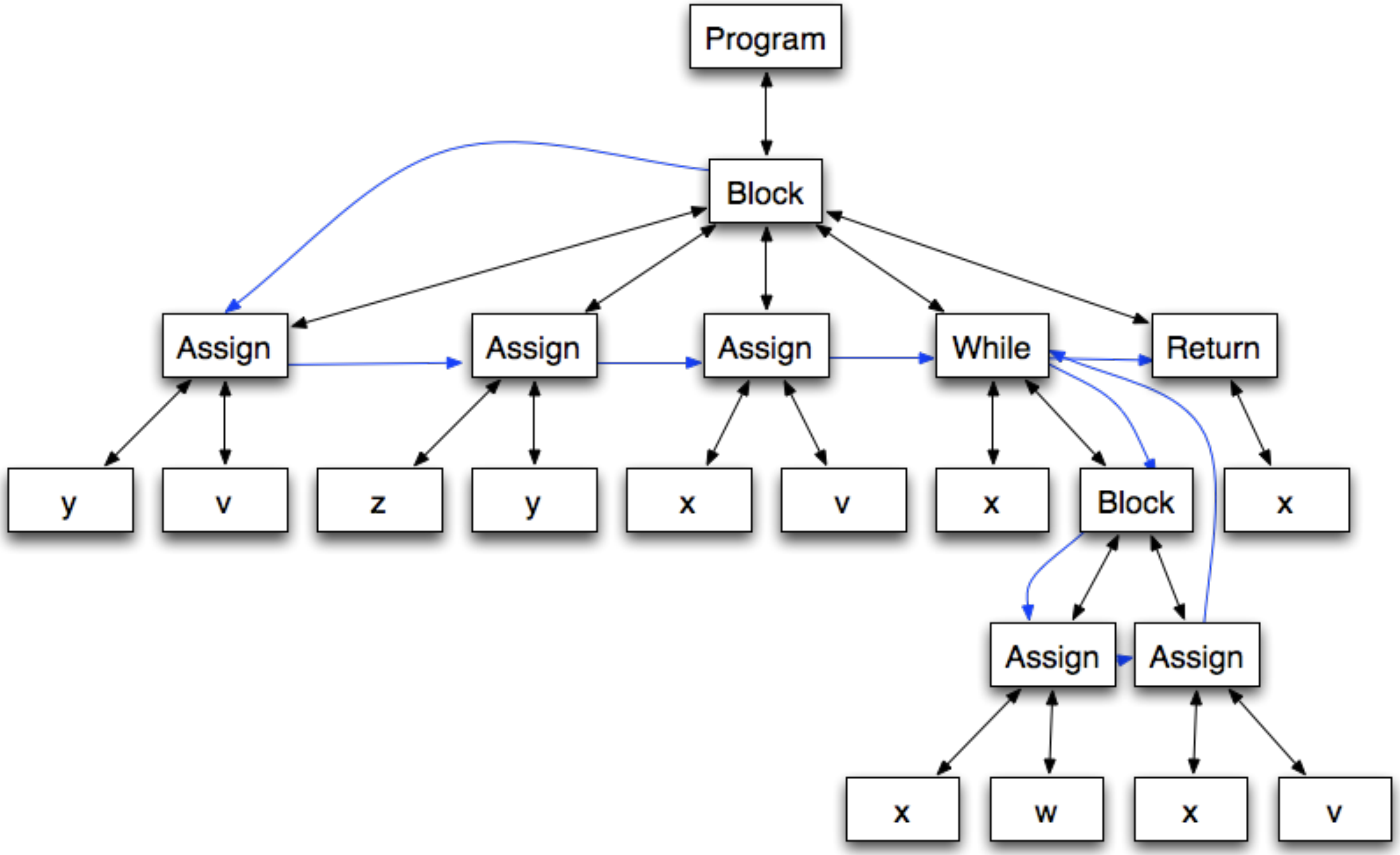
Code	*In*	*Out*
`y = v;`	{v, w}	{v, w, y}
`z = y;`	{v, w, y}	{v, w}
`x = v;`	{v, w}	{v, w, x}
`while (x)`	{v, w, x}	{v, w, x}
`{`		
`    x = w;`	{v, w}	{v, w}
`    x = v;`	{v, w}	{v, w, x}
`}`		
`return x;`	{x}	

Liveness : tree structure

```
case class Program (body : Stm) extends Attributable

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

type Var = String
```

Liveness : control flow graph

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



Liveness : successors

```
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 statements

```
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[Var] =
    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[Var] =
    attr {
        case Assign (v, _) => Set (v)
        case _             => Set ()
    }
```

Liveness : in and out dataflow equations

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

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

Liveness : in and out dataflow equations

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

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

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

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

Summary

So far, so good...

Rewriting is around 1000 lines of code, attribution around 600 lines, including comments and a largish strategy library.

Scala has proven to be a powerful and convenient basis for this work.

Ongoing activities:

- Types for strategies, attribute compositions
- Support for more language processing paradigms
- Larger use cases, performance and scalability
- Expressibility and semantics of paradigm combinations
- Correctness of semantics of paradigm hosting and combinations

Questions?

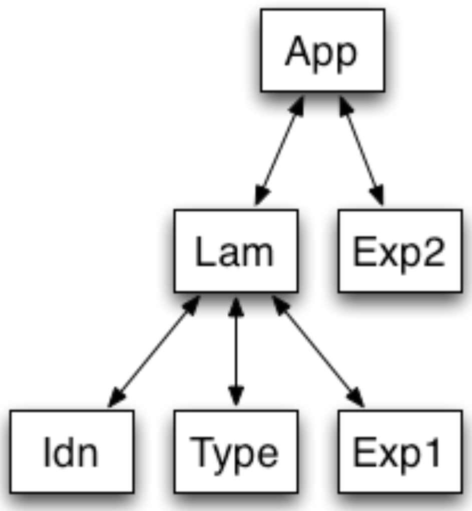
For downloads, documentation, papers, talks and mailing lists see

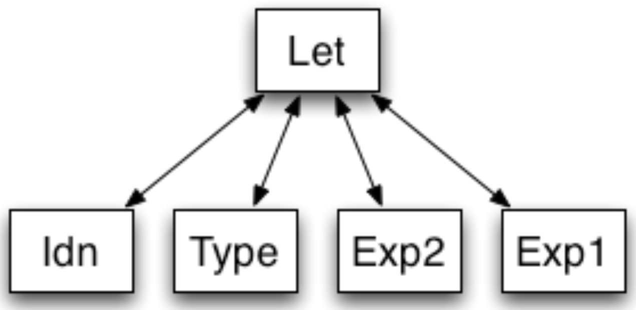
http://kiama.googlecode.com

Extras

Rewriting Rules

```
(\ x : t . e1) e2      =>      let x : t = e2 in e1
```





Rewriting Rules

```
(\ x : t . e1) e2      =>      let x : t = e2 in e1
```

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

Part 2. Rewriting

Application area: program transformation

- desugaring of high-level language constructs
- evaluation by reduction rules
- optimisation
- source to target translation

Suited for modifying the structure of the program, in contrast to attribution which usually decorates a fixed structure and is more suited to program analysis.

Part 1. Language Processing Paradigms

Formalisms and associated implementation techniques for analysing, translating and executing structured text.

- context-free grammars
- attribute grammars
- term rewriting systems

Typically realised by specific notations and tools that embody the implementation techniques.

- parser generators: YACC, JavaCC, SDF, ANTLR, Rats!, etc
- attribute grammar systems: JastAdd, Eli/LIGA, Lrc, UU-AG, etc
- term rewriting systems: Stratego, ASF+SDF, TXL, TOM, etc

Tutorial Outline

1. Kiama: motivation, aims and approach
2. Strategy-based rewriting
    - evaluation schemes for lambda calculus
3. Dynamically-scheduled attribute grammars
    - live variable analysis for imperative languages

Embedding Paradigms

Specialised notations and tools are powerful but imply overhead to

- learn paradigms and notations
- install tools and integrate with development processes
- enable multiple tools and notations to cooperate

Bring language processing paradigms closer to software developers via libraries

- use only constructs from a "general purpose" language
- what do you give up?
  - precision of notation? correctness guarantees? efficiency?

Abstract Syntax (2)

```
abstract class Type

case object IntType extends Type
case class FunType (arg : Type, res : Type)
                    extends Type

abstract class Op {
    def eval (l : Int, r : Int) : Int
}
case object AddOp extends Op { ... }
case object SubOp extends Op { ... }
```

Lifting Functions to Strategies

Scala functions can be converted to strategies.

```
def strategyf (f : Term => Option[Term]) : Strategy

val failure : Strategy =
    strategyf (_ => None)

val id : Strategy =
    strategyf (t => Some (t))
```

Queries

A query is run for its side-effects.

```
def query[T] (f : PartialFunction[Term,T]) : Strategy
```

A query to collect variable references.

```
var vars = Set[String]()
val varrefs = query { case Var (s) => vars += s }
```

(Nothing is said here about term traversal. More on that later.)

Generic Traversal

All of the strategies seen so far apply only to the current term.

The all combinator applied to a strategy s, constructs a strategy that applies s to all of the children of the current term and assembles the rewritten children under the original constructor, provided that all of the rewrites succeed.

```
def all (s : => Strategy) : Strategy
```

Similarly for some children or one child.

```
def some (s : => Strategy) : Strategy
def one (s : => Strategy) : Strategy
```

Implemented via a simple form of reflection on Scala Product types.

Name Scoping

Stratego version of strategy to look for a specific subterm:

```
issubterm =
    ?(x,y); where (<oncetd(?x)> y)
```

Kiama version:

```
val issubterm : Strategy =
    strategy {
        case (x : Term, y : Term) =>
            where (oncetd (term (x))) (y)
    }
```

Lambda Calculus with Lazy Evaluation

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

Summary

So far, so good...

Rewriting is around 1000 lines of code, including comments, library.

Scala has proven to be a powerful and convenient basis for this work.

Open issues:

- Support for more language processing paradigms in this style
- Larger use cases, performance and scalability
- Expressibility and semantics of paradigm combinations
- Correctness of semantics of paradigm hosting and combinations

Further Reading

Kiama http://kiama.googlecode.com, lambda2 example

Stratego http://strategoxt.org

Domain-Specific Language Engineering. Visser, GTTSE 2007
Program Transformation with Stratego/XT. Visser, DSPG 2004
Building Interpreters with Rewriting Strategies. Dolstra and Visser, LDTA 2002

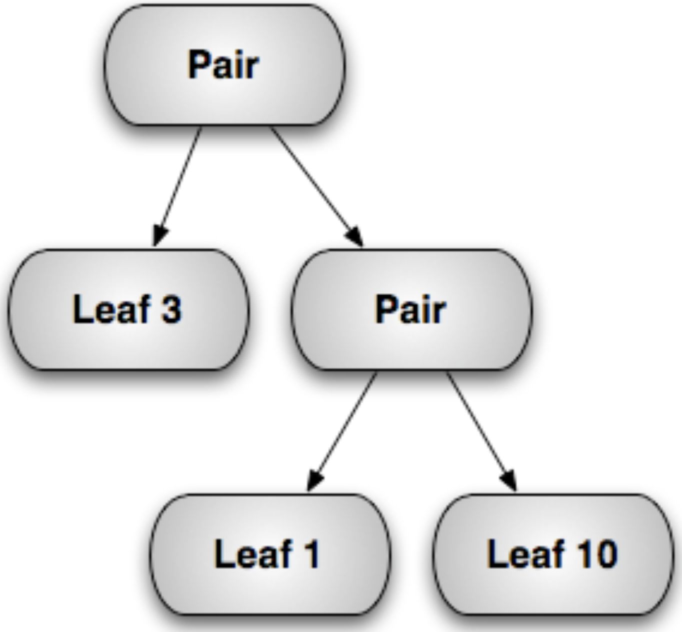
Scala http://www.scala-lang.org

Programming in Scala, Odersky. Spoon and Venners, Artima, 2008

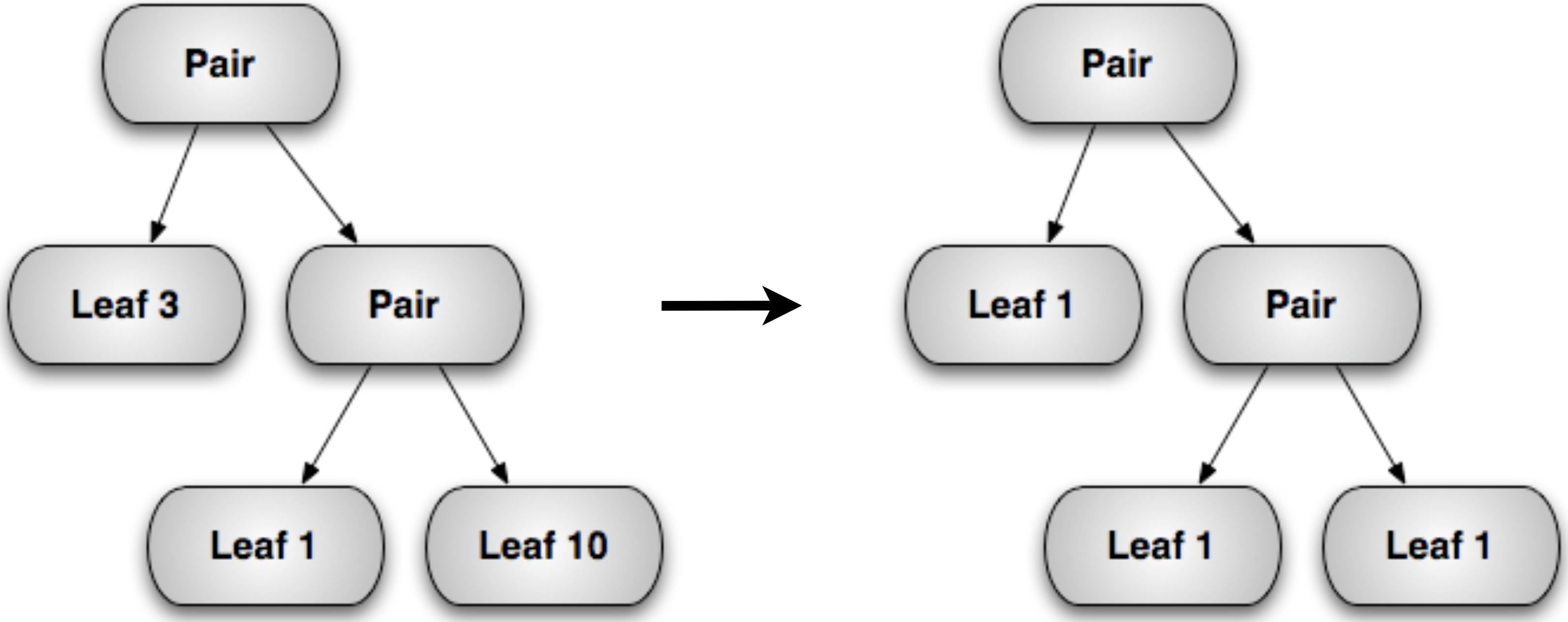
Lambda Calculus with Eager Evaluation

```
val evals : Strategy =
    attempt (traverse) <* attempt (lambda_es <* evals)

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))
    }
```

A classic example: Repmin



A classic example: 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)
  }
```

Semantic analysis

Attribute grammars are often used for analysis tasks where attributes represent semantic properties of program constructs.

Example: name and type analysis in simply-typed lambda calculus

- all uses of names should be associated with their binding occurrence
- a use without a binding occurrence is an error
- all expressions should have a type
- expressions must be used in a way that is consistent with their type

Abstract Syntax (1)

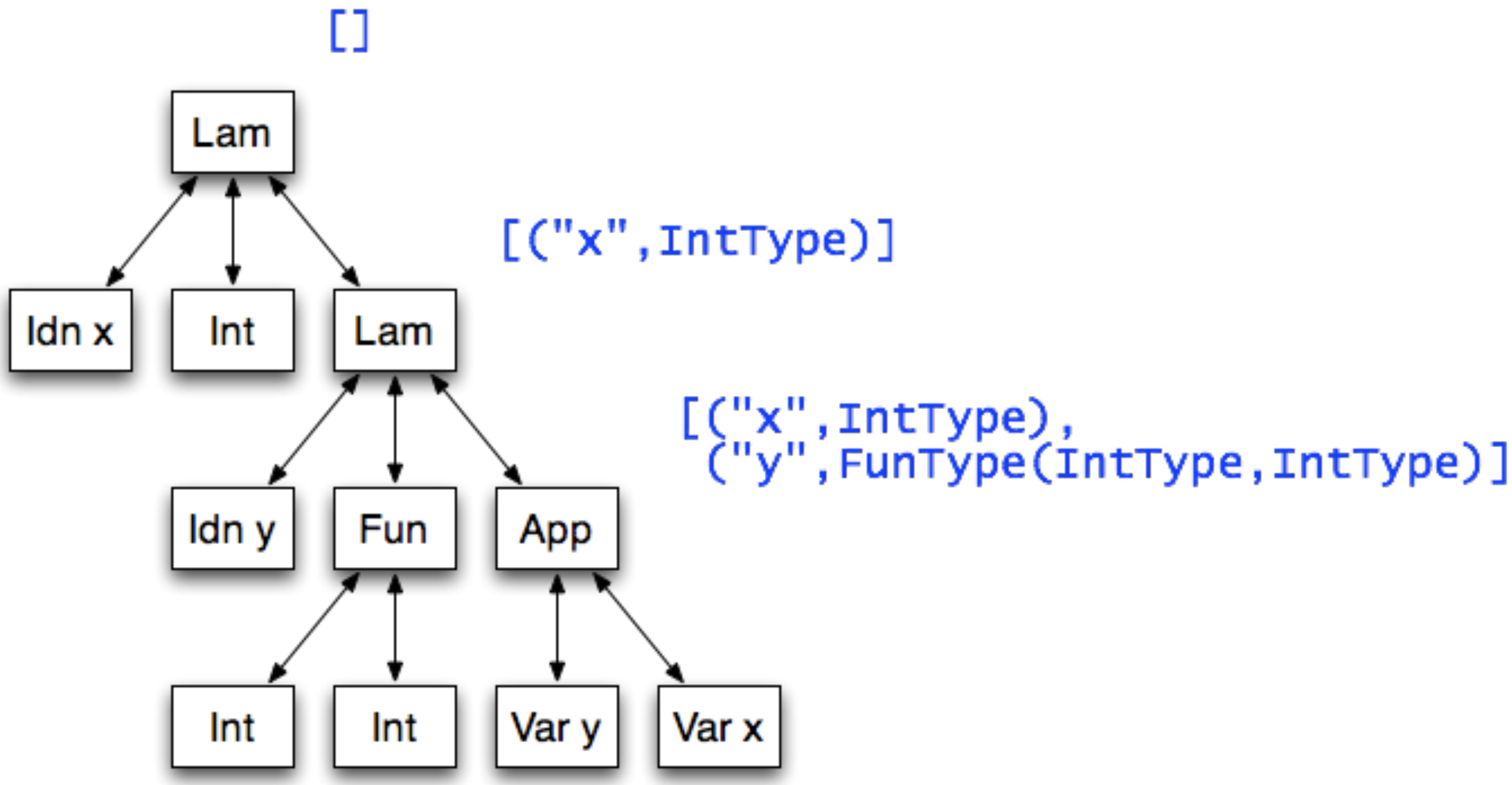
```
type Idn = String

abstract class Exp

case class Num (value : Int) extends Exp
case class Var (name : Idn) extends Exp
case class Lam (name : Idn, tipe : Type, body : Exp)
                extends Exp
case class App (l : Exp, r : Exp) extends Exp
case class Opn (op : Op, left : Exp, right : Exp)
                extends Exp
```

Abstract Syntax (2)

```
abstract class Type

case object IntType extends Type
case class FunType (arg : Type, res : Type)
                    extends Type

abstract class Op {
    def eval (l : Int, r : Int) : Int
}
case object AddOp extends Op { ... }
case object SubOp extends Op { ... }
```

Method 1: Bound variable environment

```
(\x : Int . (\y : Int -> Int . y x))
```



Method 1: Bound variable environment

```
val env : Exp ==> List[(String,Type)] =
    childAttr {
        case _ => {
            case null                  => List ()
            case p @ Lam (x, t, _) => (x,t) :: p->env
            case p : Exp               => p->env
        }
    }
```

Method 1: Defining the type of an expression (1)

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

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

      case Opn (op, e1, e2) =>
          if (e1->tipe != IntType)
              message (e1, "expected Int, found " +
                              (e1->tipe))
          if (e2->tipe != IntType)
              message (e2, "expected Int, found " +
                              (e2->tipe))
          IntType
```

Method 1: Defining the type of an expression (2)

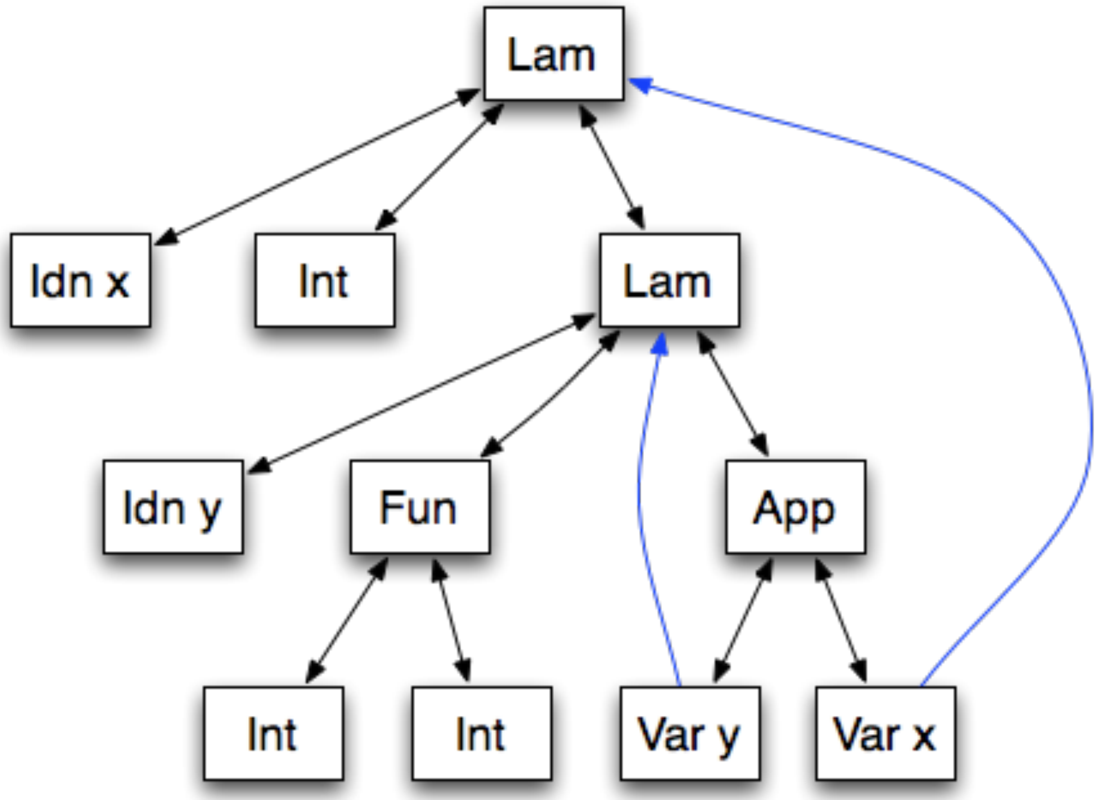
```
case App (e1, e2)      =>
    e1->tipe match {
        case FunType (t1, t2) if t1 == e2->tipe =>
            t2
        case FunType (t1, t2) =>
            message (e2, "expected " + t1 +
                         ", found " + (e2->tipe))
            IntType
        case _ =>
            message (e1, "non-function")
            IntType
    }
```

Method 1: Defining the type of an expression (3)

```
    case e @ Var (x)         =>

        (e->env).find { case (y,_) => x == y } match {

            case Some ((_, t)) => t

            case None =>
                message (e, "'" + x + "' unknown")
                IntType

        }

}
```

Method 2: Reference to binding node

```
(\x : Int . (\y : Int -> Int . y x))
```



Method 2: Reference to binding node

```
case e @ Var (x) =>

    (e->lookup (x)) match {

        case Some (Lam (_, t, _)) => t

        case None =>
            message (e, "'" + x + "' unknown")
            IntType

    }
```

Method 2: Name lookup

```
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)
    }
```

Summary

So far, so good...

Attribution is around 600 lines of code, including comments.

Scala has proven to be a powerful and convenient basis for this work.

Open issues:

- Support for more language processing paradigms in this style
- Larger use cases, performance and scalability
- Expressibility and semantics of paradigm combinations
- Correctness of semantics of paradigm hosting and combinations

Further Reading

Kiama http://kiama.googlecode.com

repmin, lambda2, dataflow examples

A Pure Object-Oriented Embedding of Attribute Grammars, Sloane, Kats, Visser, LDTA 2009

Scala http://www.scala-lang.org

Programming in Scala, Odersky. Spoon and Venners, Artima, 2008