Lightweight Language Processing in Kiama and Scala

Anthony M. Sloane

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

Anthony.Sloane@mq.edu.au

Kiama was 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

Kiama

Embedded Language Processing

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

Embedded as domain-specific languages in a general-purpose host language.

- context-free grammars
- attribute grammars
- term rewriting systems
- abstract state machines

Attribute Grammars

An equational formalism suited to specifying static properties of tree and graph structures.

Context-free grammar	Case class definitions
Attribute	Function defined on nodes
Equations	Pattern matching function definition
Evaluation mechanism	Function call, attribute caching

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}	{}

Abstract syntax

```
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 : List[Stm]) extends Stm
case class Return (ret : Var) extends Stm
case class Empty () extends Stm

type Var = String
```

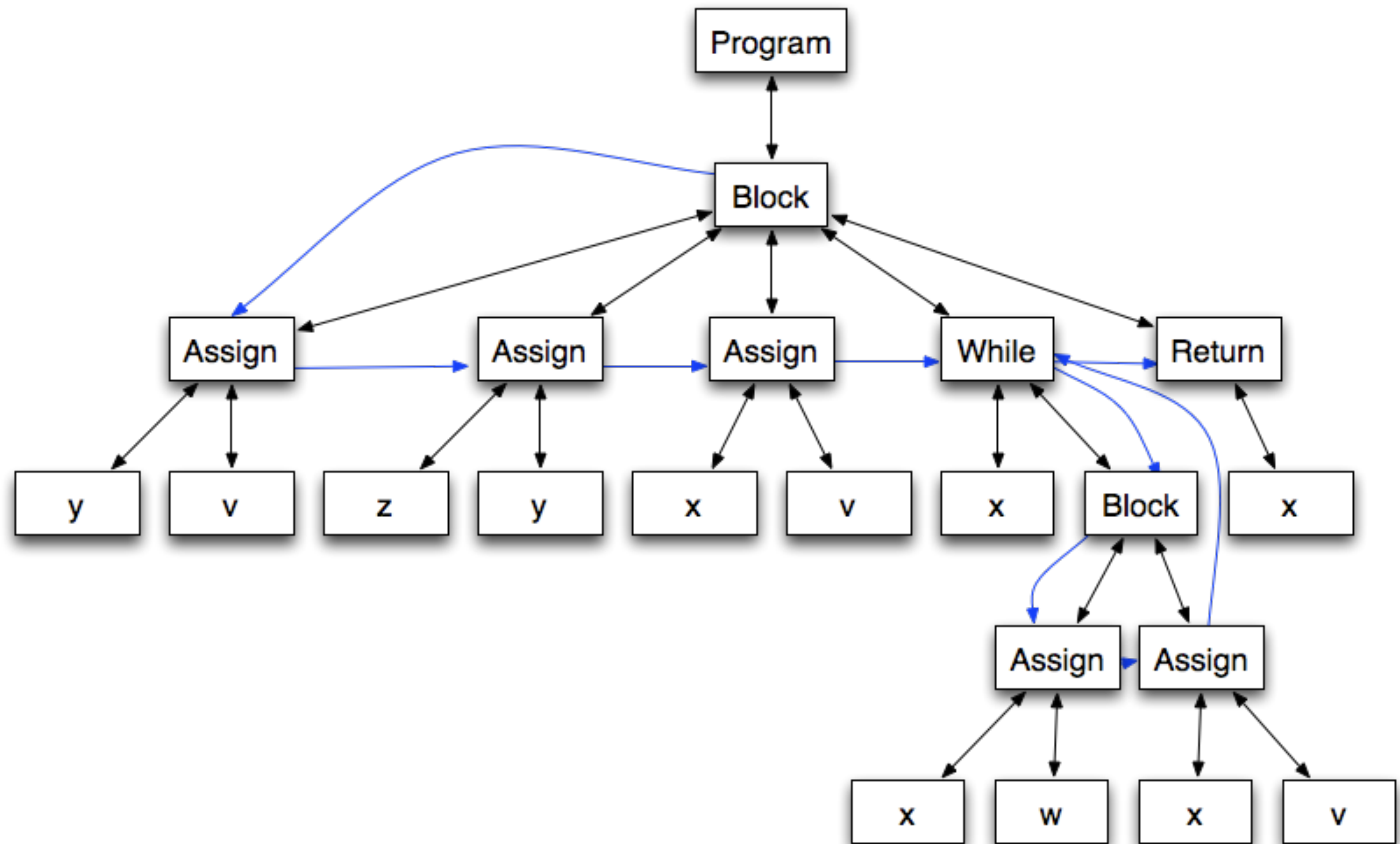
Variable uses and definitions

```
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 ()
  }
```

Control flow graph

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



Statement sequencing

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

Control flow successor

```
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
  }
```

Dataflow in and out of statements

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

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

Dataflow in and out of statements

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

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

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

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

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-based Rewriting

Rewrite rule	Pattern-matching function
Strategy	Higher-order function

A transformation of a term that either

succeeds, producing a new term, or

fails

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

Applying Strategies

A strategy is 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 fail : 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
```

Combining Strategies

Methods of 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

Generic traversal operators:

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

Dead code elimination

	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}	{}

Dead code elimination

Replace an assignment to a dead variable by an empty statement.

```
rule {
  case s @ Assign (v, _) if (! (s->out contains v)) =>
    Empty ()
}
```

Dead code elimination

Replace all dead assignment statements in a program.

```
def alltd (s : => Strategy) : Strategy =
  s <+ all (alltd (s))


val elimDeadAssign =
  alltd (
    rule {
      case s @ Assign (v, _) if (! (s->out contains v)) =>
        Empty ()
    }
  )
```

Dead code elimination

Remove empty statements and empty blocks.

```
rule {
  case Empty () :: ss => ss
  case Block (Nil)    => Empty ()
}
```

Dead code elimination

Remove empties from the bottom of the tree upwards.

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

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

val elimEmpties =
   bottomup (attempt (
     rule {
       case Empty () :: ss => ss
       case Block (Nil)    => Empty ()
     }
   ))
```

Dead code elimination

```
def optimise (t : Stm) : Stm =
  rewrite (rules) (t)

val rules = elimDeadAssign <* elimEmpties
```

Conclusion

So far, so good...

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

All hail to Scala!

Ongoing activities:

- Better types for strategies
- 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

Further information

Binary and source downloads, examples, documentation, papers, talks and mailing lists

http://kiama.googlecode.com

Next release: 1.0.0, when Scala 2.8 is released

Anthony.Sloane@mq.edu.au
http://www.comp.mq.edu.au/~asloane
Twitter: inkytonik