Embedding a Rewriting DSL in Scala

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

Overview

Strategic programming

- Stratego language

Embedding Stratego into Scala

- Rewriting in the Kiama library

Examples from Lambda Calculus evaluation

Strategic Programming

Strategic programming is generic programming using strategies.

A strategy is a generic data-processing action which can traverse into heterogeneous data structures while mixing uniform and type-specific behaviour.

The Essence of Strategic Programming
Lämmel, Visser and Visser

Our 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.

Stratego

A strategic programming language based on

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

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

http://strategoxt.org http://webdsl.org

Terms

Prefix notation

```
App (Lam ("x", IntType,
          Opn (AddOp, Var ("x"), Num (1))),
     Num (42))

Let ("x", IntType, Num (42),
     Opn (AddOp, Var ("x"), Num (1)))
```

Concrete syntax notation

```
[[ (\ x : Int -> x + 1) 42 ]]

[[ let x : Int = 42 in x + 1 ]]
```

Rewrite Rules

Evaluation of a function application

```
App (Lam (x, t, e1), e2) -> Let (x, t, e2, e1)
```

Semantics of `p -> q`

match p against the subject term
if the match succeeds,
  bind the variables x, t, e1 and e2
  build the new term q
  q is the new subject term
otherwise,
  fail

Match and Build

?p

match subject term against p

if p matches, bind any variables and succeed, leaving the subject term unchanged

if p does not match, fail

!p

build a new subject term from p, with free variables replaced by their bindings, always succeed

Combinators (1)

Identity `id` always succeed, leaving the subject term unchanged

Failure `fail` always fail

Sequential composition `p; q`

apply p to the subject term; if it succeeds, apply q to the (possibly new) subject term, otherwise fail

Guarded choice `p < q + r`

as for sequential composition, but additionally, if p fails, r is applied to the original subject term and environment

Combinators (2)

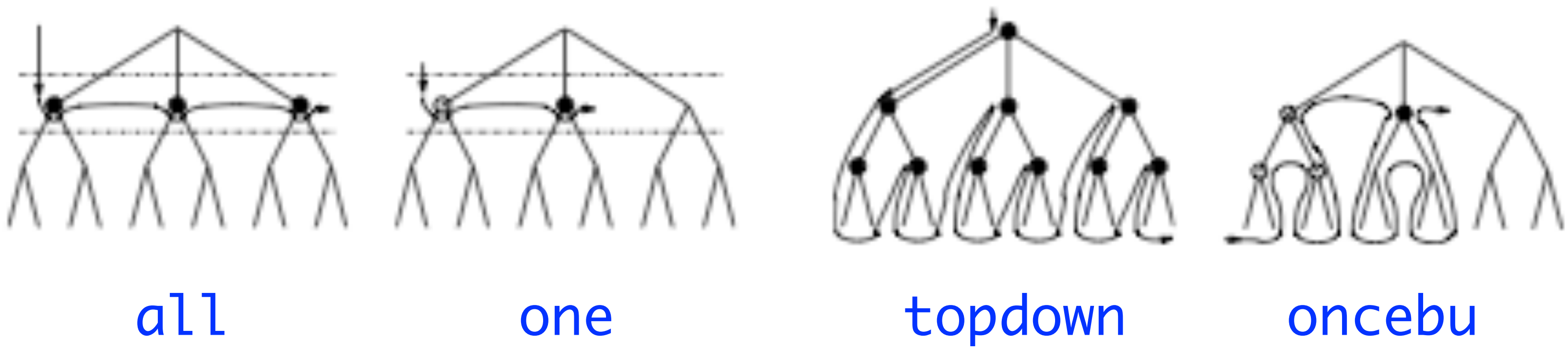
p -> q	?p; !q	rewrite rule
p <+ q	p < id + q	deterministic choice
p + q		non-deterministic choice
not (p)	p < fail + id	negation
<s> p	!p; s	application
s => p	s; ?p	binding

Note: some details of the scopes of bindings have been omitted.

Generic Traversals

The strategies seen so far apply only to the current term.

The `all`, `one` and `some` combinators applied to a strategy `s`, construct strategies that apply `s` to all, one or some of the children of the current term and assemble the rewritten children under the original constructor, provided that the rewrites succeed.



(from *The Essence of Strategic Programming*)

Strategy library examples

```
topdown (s) = s; all (topdown (s))

oncebu (s) = one (oncebu (s) <+ s)

oncetd (s) = s <+ one (oncetd (s))

beloweq (s, t) = oncetd (t; oncetd (s))

untileq (s, t) = s; t <+ one (untileq (s, t))
```

The Kiama Library

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

Paradigms supported at present:

- strategy-based term rewriting (this talk)
- dynamically-scheduled attribute grammars
- abstract state machines (in progress)

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

http://www.scala-lang.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]
```

`Term` is anything that implements the `Product` interface (needed for generic traversals).

Lambda Calculus Term 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))
```

Combining Strategies

Methods of the Strategy class allow strategies to be combined.

`p <* q`	sequence
`p < q + r`	guarded choice
`p <+ q`	deterministic 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)`

Combinator Implementation

```
abstract class Strategy ... { p =>

  def apply (r : Term) : Option[Term]

  def <* (q : => Strategy) : Strategy =
      new Strategy {
          def apply (t1 : Term) =
              p (t1) match {
                  case Some (t2) => q (t2)
                  case None      => None
              }
      }
  ...
}
```

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)
```

Lifting to Strategies

Function values can be usefully lifted to strategies.

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

val failure = strategyf (_ => None)
val id = strategyf (t => Some (t))
```

Implicit lifting for common cases.

```
implicit def termToStrategy (t : Term) =
    strategyf (_ => Some (t))

implicit def optionToStrategy (o : Option[Term]) =
    strategyf (_ => o)
```

Rewrite Rules

Rewrite rules are defined by Scala partial functions.

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

Beta reduction using Scala's case syntax for partial functions.

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

More Rewriting Rules

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


def term (t : Term) =
    rule {
        case `t` => t
    }
```

Queries

A query is run for its side-effects.

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

A query to collect variable references.

```
def variables (e : Exp) : Set[String] = {
    var vars = Set[String]()
    everywheretd (query {
                            case Var (s) => vars += s
                        }) (e)
    vars
}
```

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)
    }
```

Library Strategies

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

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)
}
```

Lambda Calculus with Meta-level Substitution

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

val s = 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 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)
    }
```

Explicit Substitution

```
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)))
    }
```

Congruences (work in progress)

Apply strategies to the components of a particular term structure.

Stratego

```
App (s1, s2)
```

Kiama:

```
AppC (s1, s2)

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

Eager and Lazy Evaluation

Eager

```
val s : Strategy =
    attempt (AppC (s, s) + LetC (id, id, s, s) +
             OpnC (id, s, s)) <*
    attempt (lambda <* s)
```

Lazy (no sharing)

```
val s : Strategy =
    attempt (AppC (s, id) + LetC (id, id, id, s) +
             OpnC (id, s, s)) <*
    attempt (lambda <* s)
```

Conclusion

The rewriting part of Kiama is around 1000 lines of Scala code, including comments and a largish strategy library.

The experiment shows the clear tradeoff between the lightweight nature of embedding vs analysis and optimisation opportunities from a separate language.

Ongoing activities:

- Congruences
- Types for strategies
- Larger use cases, performance and scalability
- Concrete syntax
- Correctness of semantics of embedding

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

Extras

Explicit Substitutions

```
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))
    }
```