Tom

Strategic rewriting in Java

Programming with rewrite rules

- Advantages
  - matching is an expressive mechanism
  - rules express elementary transformations
- Limitations
  - rewrite systems are often non-terminating and non-confluent
  - usually, we don't want to (try to) apply all the rules in the same time

Example of non-terminating rewrite system

$(x+y)*z \rightarrow (x*z)+(y*z)$

$z*(x+y) \rightarrow (z*x)+(z*y)$

$(x*z)+(y*z) \rightarrow (x+y)*z$

$(z*x)+(z*y) \rightarrow z*(x+y)$

Control rule application

- Classical solution
  - introduce a new operator f to reduce the set of rules used for normalization
  - $l \rightarrow r$ becomes $f(l) \rightarrow r'$
  - operator f is used to control the rules to be applied

Encoding the control

distrib((x+y)*z) → (x*z)+(y*z)

distrib(z*(x+y)) → (z*x)+(z*y)

facto((x*z)+(y*z)) → (x+y)*z

facto((z*x)+(z*y)) → z*(x+y)

- starting from a term t, we can repeat the reduction t = distrib(t) until a fix-point is obtained and then factorize with t = facto(t)

Example

- Reduce t = a + ((a+b)*c)
  - distrib(t) = ?
- Add new rules to propagate the application of the rules
  - distrib(x+y) = distrib(x) + distrib(y)
  - distrib(x*y) = distrib(x) * distrib(y)
  - etc.

Consequences

- The congruence should be defined explicitly for each rule and each constructor
- The fix-point should be defined explicitly
- There is no separation between the transformation and the control and thus,
  - more difficult to understand
  - rules are not reusable

What we would like

- control rule application by
  - specifying the “traversal” of a term (i.e. apply the rules on the sub-terms)
  - keeping separate the rules and the control (strategy)

Solution

- Use strategies
- Combine elementary transformations
- Examples

  dnf = innermost(DL <+ DR <+ ...)

  DL : (x+y)*z → (x*z)+(y*z)

  DR : z*(x+y) → (z*x)+(z*y)

Elementary strategies

Rewrite rule

- A rule $R : g \rightarrow d$ is an elementary strategy
- Examples : $R = a \rightarrow b$
  - $(R)[a] = b$
  - $(R)[b] = fail$
  - $(R)[f(a)] = fail$

Identity and failure

- id: does nothing but doesn't fail
- fail: fails all the time
- Examples
  - (id)[a] = a
  - (id)[b] = b
  - (fail)[a] = fail

Composition

- S1 ; S2
- Apply S1, then S2
- Fails if S1 or S2 fail
- Examples
  - (a → b ; b → c)[a] = c
  - (a → b ; c → d)[a] = fail
  - (b → c ; a → b)[a] = fail

Choice

- S1 <+ S2
- Apply S1. If it fails, apply S2
- Examples
  - (a → b <+ b → c)[a] = b
  - (b → c <+ a → b)[a] = b
  - (b → c <+ c → d)[a] = fail
  - (b → c <+ id)[a] = a

Some equivalent strategies

- id ; s = s
- s ; id = s
- id <+ s = id
- fail <+ s = s
- s <+ fail = s
- fail ; s = fail
- s ; fail ≠ fail (pourquoi ?)

Advanced strategies

- try(s) = s <+ id
- repeat(s) = try(s ; repeat(s))
- Examples
  - (try(b → c))[a] = a
  - (repeat(a → b))[a] = b
  - (repeat(b → c <+ a → b))[a] = c
  - (repeat(b → c))[a] = a

Traversal primitives

- apply a strategy to one or several direct descendants
  - congruence
    - apply a (different) strategy to each descendant of a constructor
  - all
    - apply a strategy to all descendants
  - one
    - apply a strategy to one descendant

Congruence

- $c(S_1,\ldots,S_n)$ for each constructor c
- Examples
  - (f(a → b))[a] = fail
  - (f(a → b))[f(a)] = f(b)
  - (f(a → b))[f(b)] = fail
  - (g(try(b → c) , try(a → b)))[g(a,a)] = g(a,b)
- Exercise
  - define the strategy map for the lists built on (cons,nil)

Generic congruence

- all(S), fails if S fails on one of the descendants
- Application on constant: all(S)[cst] = cst
- Examples
  - (all(a → b))[f(a)] = f(b)
  - (all(a → b))[g(a,a)] = g(b,b)
  - (all(a → b))[g(a,b)] = fail
  - (all(a → b))[a] = a
  - (all(try(a → b)))[g(a,c)] = g(b,c)

Generic congruence

- one(S), fails if S cannot be applied at least on one of the descendants
- Application on constant: one(S)[cst] = fail
- Examples
  - (one(a → b))[f(a)] = f(b)
  - (one(a → b))[g(a,a)] = g(a,b)
  - (one(a → b))[g(b,a)] = g(b,b)
  - (one(a → b))[a] = fail

Traversal strategies

- oncebu(S) = one(oncebu(S)) <+ S
- oncetd(S) = S <+ one(oncetd(S))
- innermost(S) = repeat(oncebu(S))
- outermost(S) = repeat(oncetd(outermost(S)))
- bottomup(S) = all(bottomup(S)) ; S
- topdown(S) = S ; all(topdown(S))
- innermost(S) = bottomup(try(S ; innermost(S)))

Strategies in Tom

Elementary constructions

- Identity
- Fail
- Sequence
- Choice
- All
- One
- mu

Utilisation

- A strategy has type Strategy
  - Strategy s = `Identity();
- a term is Visitable (i.e. implements the interface)
  - Visitable t = `a();
- A strategy can be applied on a term
  - result = s.visit(t);
- A strategy preserves the type
  - Term t = `a();
  - Term result = (Term) s.visit(t);

Elementary strategy in TOM

```
%strategy RewriteSystem extends Fail() {
  visit Term {
    a() -> b()
  }
}
```

Strategy definition

```
Strategy Try(Strategy S) {
  return `Choice(S,Identity())
}
Strategy Repeat(Strategy S) {
  return `mu(MuVar("x"),Choice(Sequence(S,MuVar("x")),Identity()));
}
Strategy OnceBottomUp(Strategy S) {
  return `mu(MuVar("x"),Choice(One(MuVar("x")),S));
}
```

- Exercise : implement innermost(a →b)

Examples

```
Strategy rule = new RewriteSystem();

Term subject = `f(g(g(a,b),g(a,a)));

`OnceBottomUp(rule).visit(subject);

`Innermost(rule).visit(subject);

`Repeat(OnceBottomUp(rule)).visit(subject);
```

InnerMost

```
Strategy rule = new RewriteSystem();

Term subject = `f(g(g(a,b),g(a,a)));

Strategy innermost =
   `mu(MuVar("x"),Sequence(All(MuVar("x")),Choice(Sequen
   ce(rule,MuVar("x")),Identity)));

innermost.visit(subject));
```

Question

- How to compute result sets
- Example
  - f(g(g(a,b),g(a,b)))
  - find x such that g(x,b) matches a sub-term

Solution

- Consider

  s(col) : g(x,b) → col.add(x)

- Apply

  Try(BottomUp(s(col)))

- Enumerate col

Codage

```
%strategy RewriteSystem(c:Collection) extends Identity() {
   visit Term {
      g(x,b()) -> { collection.add(`x); }
   }
}
Collection collection = new HashSet();
Strategy rule = `RewriteSystem(collection);
Term subject = `f(g(g(a,b),g(c,b)));
`Try(BottomUp(rule)).visit(subject);
System.out.println("collect : " + collection);
```

Codage

```
%strategy RewriteSystem(c:Collection) extends Identity() {
   visit Term {
      g(x,b()) -> { collection.add(`x); }
   }
}
Collection collection = new HashSet();
Strategy rule = `RewriteSystem(collection);
Term subject = `f(g(g(a,b),g(c,b)));
`Try(BottomUp(rule)).visit(subject);
System.out.println("collect : " + collection);
```

Program optimization

```
%gom {
  module Term
  Bool = True()
        | False()
        | Neg(b:Bool)
        | Or(b1:Bool, b2:Bool)
        | And(b1:Bool, b2:Bool)
        | Eq(e1:Expr, e2:Expr)
  Expr = Var(name:String)
      | Cst(val:int)
      | Let(name:String, e:Expr, body:Expr)
      | Seq( Expr* )
      | If(cond:Bool, e1:Expr, e2:Expr)
      | Print(e:Expr)
      | Plus(e1:Expr, e2:Expr)
```

If(Neg(b),i1,i2)->If(b,i2,i1)

```
public Expr optiIf(Expr expr) {
        %match(Expr expr) {
            If(Neg(b),i1,i2) -> { return `opti(If(b,i2,i1)); }
            x -> { return `x; }
        }
        thro
}

Expr
    If(Ne
        Se
```

```
p4 =
Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
Seq(Print(Var("i")),Let("i",Plus(Var("i"),Cst(1)),Var("i"))),Seq()))
OPTI p4 =
Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
Seq(Print(Var("i")),Let("i",Plus(Var("i"),Cst(1)),Var("i"))),Seq()))
```

```
System.out.println("p4 = \n" + p4);
System.out.println("OPTI p4 = \n" + optiIf(p4));
```

If(Neg(b),i1,i2)->If(b,i2,i1)

```
public static Expr optiNaive(Expr expr) {
  %match(expr) {
    If(Neg(b)
    //
  }
  th
}
```

```
p4 =
Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
Seq(Print(Var("i")),Let("i",Plus(Var("i"),Cst(1)),Var("i"))),Seq()))
OPTI p4 =
Let("i",Cst(0),If(Eq(Var("i"),Cst(10)),Seq(),
Seq(Print(Var("i")),Let("i",Plus(Var("i"),Cst(1)),Var("i")))))
```

Can we use strategies?

Simpler problem:

- Find all constants in a program

```
%strategy stratPrintCst() extends `Fail() {
    visit Expr {
        Cst(x) -> { System.out.println("cst: " + `x); }
    }
}
```

```
`TopDown(Try(stratPrintCst())).visit(p4);
```

```
cst: 0
cst: 10
cst: 1
```

Another solution

```
%strategy FindCst() extends `Fail() {
    visit Expr {
        c@Cst(x) -> { return `c; }
    }
}
%strategy PrintTree() extends `Identity() {
    visit Expr {
        x -> { System.out.println(`x); }
    }
}

`TopDown(Try(Sequence(FindCst(),PrintTree()))).visitLight(expr);
```

```
Cst(0)
Cst(10)
Cst(1)
```

Back to the optimizer

```
public Expr optiIf(Expr expr) {
        %match(Expr expr) {
            If(Neg(b),i1,i2) -> { return `opti(If(b,i2,i1)); }
            x -> { return `x; }
        }
        throw new RuntimeException("strange term: " + expr);
}
```

```
    %strategy OptIf() extends `Fail() {
        visit Expr {
            If(Neg(b),i1,i2) -> { return `If(b,i2,i1); }
        }
    }
```

Back to the optimizer

```
%strategy OptIf() extends `Fail() {
  visit Expr {
     If(Neg(b),i1,i2) -> { return `If(b,i2,i1); }
  }
}
```

```
System.out.println("OPTI p4 = \n" + `Innermost(OptIf()).visit(p4));
```

```
`Sequence(Innermost(OptIf()),PrintTree()).visitLight(p4);
```

What do we have ?

- efficient data-structures (maximal sharing)
- rewrite rules (labeled and unlabeled)
- strategies (congruence, parameterized, etc.)
- ∅, A, and AU matching (non-linear)
- anti-patterns: !conc(_*,a(),_*)
- everything, smoothly integrated into Java