Strategic Programming in Java

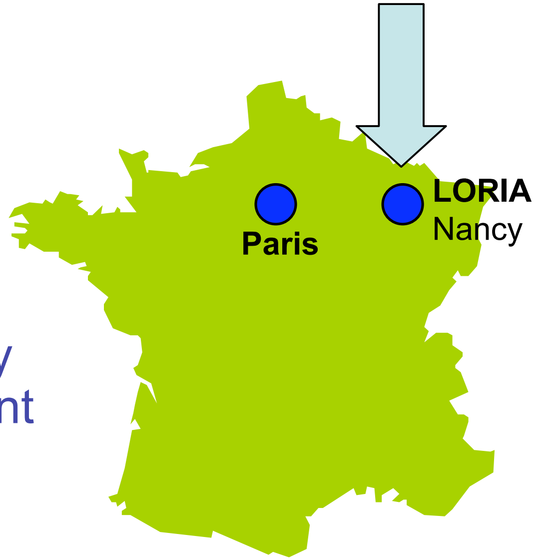
Pierre-Etienne Moreau
Antoine Reilles

Software Transformation Systems Workshop
October, 22nd, 2006

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA

Motivations

- Rule Based Programming is a nice idea !
- We interested in promoting and integrating concepts and tools in existing environments:
  - ASF+SDF, ELAN, Maude, Stratego, TXL are very nice, but difficult to use in a C or Java environment
- Our approach:
  - take the best of these languages
  - shake, shake, shake
  - design and develop a set of Java tools that offer similar constructs:
    - algebraic data-type (Gom)
    - equational pattern matching (Tom)
    - strategic programming (Strategy Library)
  - note that something is missing: concrete syntax



Tom in five minutes

- A Java program is a Tom program

```
import pil.term.types.*;
import java.util.*;
import jjtraveler.VisitFailure;
import jjtraveler.reflective.VisitableVisitor;
public class Pil {
  …
  public final static void main(String[] args) {
    Expr p1 = …;
    System.out.println("p1 = " + p1);
    …
  }
}
```

Tom adds algebraic data-types to Java

- Gom supports many-sorted first order signature

```
import pil.term.types.*;
import java.util.*;
import jjtraveler.VisitFailure;
import jjtraveler.reflective.VisitableVisitor;
public class Pil {
  …
  public final static void main(String[] args) {
    Expr p1 = …;
    System.out.println("p1 = " + p1);
    …
  }
}
```

```
%gom {
  module Term
    imports int String
    abstract syntax
    Bool =
          | True()
          | False()
          | Eq(e1:Expr, e2:Expr)
    Expr =
          | Var(name:String)
          | Let(var:Expr, e:Expr, body:Expr)
          | Seq(i1:Expr, i2:Expr)
          | If(cond:Bool, e1:Expr, e2:Expr)
          | a()
          | b()
}
```

An algebraic term is a Java object

- Back-quote (`) to build a term

```
import pil.term.types.*;
import java.util.*;
import jjtraveler.VisitFailure;
import jjtraveler.reflective.VisitableVisitor;
public class Pil {
  …
  public final static void main(String[] args) {
    Expr p1 = `Let(Var("x"),a(), Let(Var("y"),b(),Var("x")));
    System.out.println("p1 = " + p1);
    …
  }
}
```

```
%gom {
  module Term
    imports int String
    abstract syntax
    Bool =
           | True()
           | False()
           | Eq(e1:Expr, e2:Expr)
    Expr =
           | Var(name:String)
           | Let(var:Expr, e:Expr, body:Expr)
           | Seq(i1:Expr, i2:Expr)
           | If(cond:Bool, e1:Expr, e2:Expr)
           | a()
           | b()
}
```

INRIA

Tom adds pattern matching to Java

- %match supports syntactic and associative pattern matching

```
import pil.term.types.*;
import java.util.*;
import jjtraveler.VisitFailure;
import jjtraveler.reflective.VisitableVisitor;
public class Pil {
  …
  public final static void main(String[] args) {
    Expr p1 = …;
    System.out.println("p1 = " + p1);
    …(pretty(p1));
  }
  …
}
```

```
public static String pretty(Object o) {
   %match(o) {
     Var(name) -> { return `name; }

     Let(var,expr,body) -> {
       return "let " + pretty(`var) + "<-" + pretty(`expr) +
              " in " + pretty(`body);
     }

     Seq(i1,i2) -> { return pretty(`i1) + " ; " + pretty(`i2); }

     If(c,i1,i2) -> { return "if(" + pretty(`c) + ") " +
                             pretty(`i1) + " else " + pretty(`i2) + " end"; }

     Eq(e1,e2) -> { return pretty(`e1) + " = " + pretty(`e2); }

   }
   return o.toString();
 }
```

Summary

- Tom offers 3 new constructs:
  - %gom
  - `
  - %match
- This is powerful, but clearly not enough
- There is no separation between Transformation and Control
- Question:
  - starting from the JJTraveler library (J. Visser, OOPSLA 2001)
  - studying ASF+SDF, ELAN, and Stratego
  - can we design a powerful strategy language, usable in Java ?
- Shake, shake, shake… the answer is Yes

Elementary strategies

- Identity and Fail are elementary strategies
- A Rule is an elementary strategy

```
%strategy RenameVar(n1:String,n2:String) extends Identity() {
   visit Expr {
     Var(n) -> { if(`n==n1) return `Var(n2); }
   }
 }
```

```
Expr p1 = `Let(Var("x"),a(), Let(Var("y"),b(),Var("x")));

Expr p2 = `RenameVar("x","z").apply(p1);

> Let(Var("x"),a(), Let(Var("y"),b(),Var("x")))
```

- a strategy is built using `
- “x” and “z” are parameters of sort String
- the rule is applied once, at root position

Basic strategies

- Similarly to Stratego and JJTraveler we consider:
  - Sequence, Choice, All, One, Not, …
  - to build more complex strategies: parameterized and recursive
    - Try(s) = Choice(s,Identity)
    - Repeat(s) = Try(Sequence(s,Repeat(s)))
    - BottomUp(s) = Sequence(All(BottomUp(s)),s)

```
Expr p1 = `Let(Var("x"),a(), Let(Var("y"),b(),Var("x")));

Expr p2 = `BottomUp(RenameVar("x","z”)).apply(p1);

> Let(Var(”z"),a(), Let(Var("y"),b(),Var(”z")))
```

- Big difference: BottomUp is user defined using the mu operator

```
public Strategy BottomUp(Strategy s) {
  return `mu(MuVar("x"),Sequence(All(MuVar("x")),s));
}
```

- Note: a strategy is a term, that can be matched, traversed, etc.

Parameterized strategies

- a strategy can be parameterized by values:

```
%strategy RenameVar(n1:String,n2:String) extends Identity() { … }
```

- a strategy can do side effects:

```
%strategy CollectVar(c:Collection) extends Identity() {
   visit Expr {
     v@Var(_) -> { c.add(v) }
   }
 }
Collection set = new HashSet();
`BottomUp(CollectVar(set)).apply(p1);
```

Strategies parameterized by a strategy

- sometimes we need to recursively call the current calling strategy

```
`BottomUp(Rule()).visit()

%strategy Rule() extends Identity() {
    visit Expr {
      Let(v,e,body) -> { … `BottomUp(Rule()).apply(body); … }
    }
  }
```

- this breaks separation between rules and control
- solution: give the calling context as argument

```
`mu(MuVar("s"),BottomUp(Rule(MuVar("s")).visit()

%strategy Rule(s:Strategy) extends Identity() {
    visit Expr {
      Let(v,e,body) -> { … `s.apply(body); }
    }
  }
```

Another big news

- a strategy knows its context: the position where it is applied

```
%strategy CollectVar(c:Collection) extends Identity() {
   visit Expr {
     Var(_) -> { c.add(getPosition()) }
   }
 }
Collection set = new HashSet();
`BottomUp(CollectVar(set)).apply(p1);
```

- a position is a list of integers that can become strategy:
  - pos=1.2.1 leads to Omega(1,Omega(2,Omega(1,s)))
- very useful:
  - to perform non-deterministic computations
  - to check global properties (collect positive variables for example)

To conclude

- we now have a powerful framework to
  - define algebraic data-types in Java
  - maintain them in canonical forms (thanks to Gom)
  - define transformations (thanks to associative pattern matching)
  - control and reuse them (thanks to strategies)
- fully integrated into Java
- used in several applications
  - the Tom compiler itself
  - bytecode analysis and transformation framework
  - proof assistant for supernatural deduction (proof manipulation)
- current and further work
  - a strategy library for graph traversal and transformation
  - support for concrete syntax

Tom is available at: tom.loria.fr