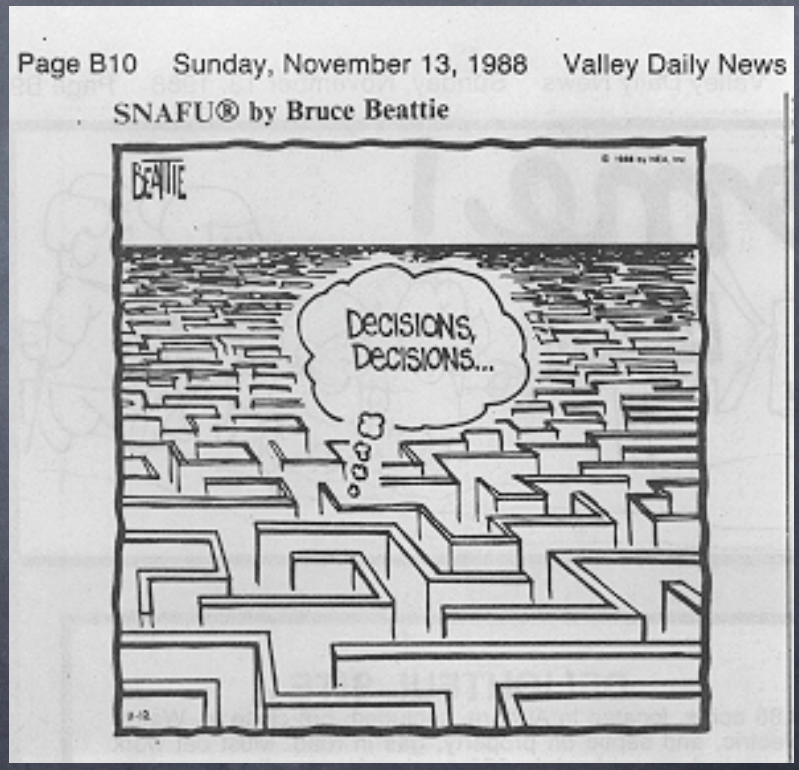
Tom: Piggybacking Rewriting on Java

Horatiu Cirstea
Pierre-Etienne Moreau

Vocabulary

- Rule: something that performs an action, that transforms a data
  - ex: Move N, S, E, W
- Strategy: something that explores, that controls how the rules are applied
  - ex: random, right-wall



Find the exit

An example


+33* → France
+333* → Est
+33383* → D.54
+0800* → Gratuit
+33383593019
France, Est, D.54

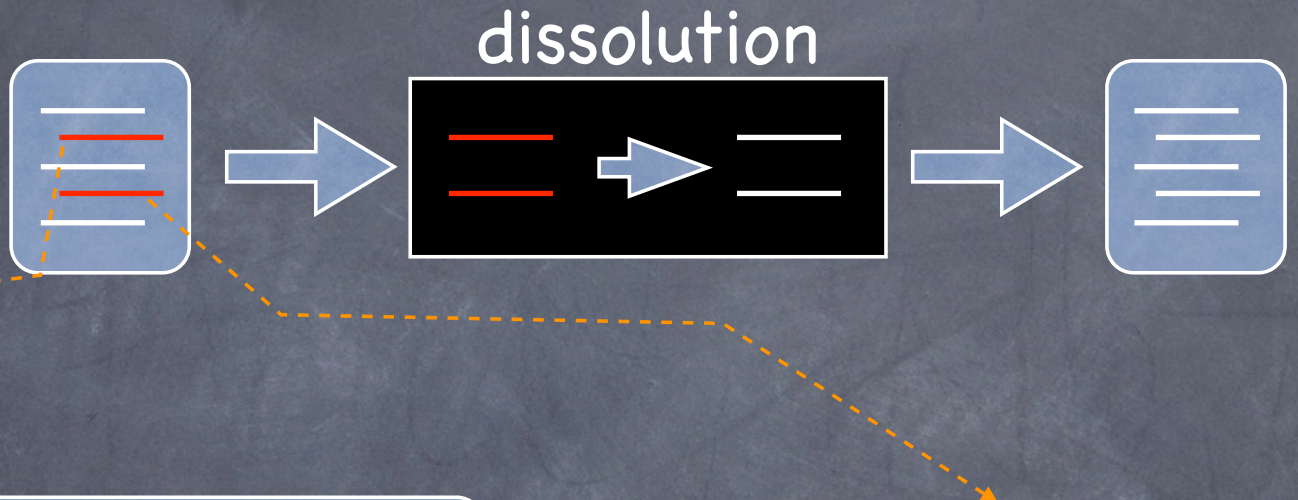
Rule based languages

- there exists several implementations of term rewriting
  - ASF+SDF, CafeOBJ, ELAN, Maude, OBJ, Stratego, TL, TXL, etc.
- we present Tom
  - an extension of Java
  - Tom = Java + terms + rules + strategies

TOM

- A language that allows the programmer:
  - to build tree structures
  - find patterns in a data structure
- When a pattern is found
  - we say that the pattern «matches» against the data structure
  - we can execute an action, written in C or Java

TOM compilation




```
Nat   = zero()
      | suc(Nat)
      | plus(Nat,Nat)
```

```
plus(x,zero()) -> x
plus(x,suc(y)) -> suc(plus(x,y))
```

Corba, ...)

TOM

Main structure: terms

- Given a many-sorted signature
- Example : Peano integers
  - zero : -> Nat
  - suc : Nat -> Nat

Gom can be used to define the signature

```
%gom {
    module Peano
    abstract syntax
    Nat  = zero()
         | suc(n:Nat)
}
```

Basic mechanism: the ` operator

- used to build terms
  - Example: `suc(suc(zero()))

The term should be well-formed and well-typed

Simple TOM program

```
public class Peano{
   %gom{
        module peano
        abstract syntax
        Nat =   zero()
              | suc(n:Nat)
   }
   public static void main(String[] args){
      System.out.println("One="+`suc(zero()));
   }
}
```

The program prints:

```
suc(zero())
```

Don't forget the `

```
System.out.println("One="+ suc(zero()));
```

- generates an error since `suc` is not a Java method

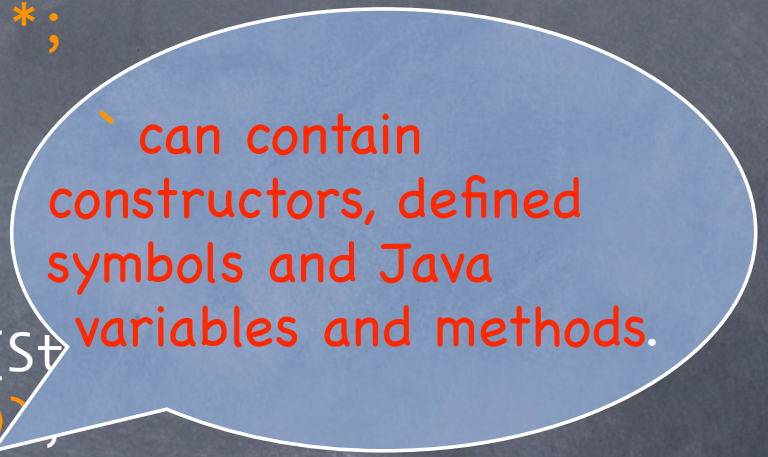
Java variables with TOM sorts

```
import peano.peano.types.*;

public class Peano{
 %gom{ ...}

 public static void main(St
    Nat one = `suc(zero());
    Nat two = `suc(one);

    System.out.println("one="+two);
 }
}
```



The program prints:

```
suc(suc(zero()))
```

Rewrite rules

```
%gom {
    module Peano
    abstract syntax
    Nat   = zero()
          | suc(n:Nat)
          | one()
          | two()
     module Peano:rules() {
       one() -> suc(zero())
       two() -> suc(one())
     }
}
```

Rule application

```
> System.out.println(`one())
    suc(zero())
> System.out.println(`two())
    suc(suc(zero()))
```

Implement addition with rewrite rules

```
module Peano:rules() {
    one() -> suc(zero())
    two() -> suc(one())
    plus(x,zero()) -> x
    plus(x,suc(y)) -> suc(plus(x,y))
 }
```

> System.out.println(`plus(one(),two()))

> Remark: variable types are inferred

Exercise

Write the TOM program that computes the Fibonacci numbers

- fib(0) = 1
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2) for all $n \geq 2$

Rewrite system

```
Nat  = ... | fib(n:Nat)
module Peano:rules() {
        one() -> suc(zero())
        plus(x,zero()) -> x
        plus(x,suc(y)) -> suc(plus(x,y))
        fib(zero()) -> one()
        fib(suc(zero())) -> one()
        fib(suc(suc(x))) -> plus(fib(x),fib(suc(x)))
  }
```

Question

Can we write the following program ?

```
module Peano:rules() {
    fib(zero()) -> one()
    fib(one()) -> one()
    fib(suc(suc(x))) -> plus(fib(x),fib(suc(x)))
}
```

Exercise

- Evaluate the term fib(plus(one(),two()))

```
one()
-> suc(zero())
two()
-> suc(suc(zero()))
plus(suc(zero()),suc(suc(zero())))
-> suc(plus(suc(zero()),suc(zero())))
-> suc(suc(plus(suc(zero()),zero())))
-> suc(suc(suc(zero())))
fib(suc(suc(suc(zero()))))
-> plus(fib(suc(zero())),fib(suc(suc(zero()))))
fib(suc(zero()))
-> ?
-> fib(suc(zero()))
```

Résumé

- %gom can be used to define signatures
- rules() can be used to define a rewrite system containing:
  - constructors and variables below the top symbol of a lhs
  - constructors, variables and defined symbols in the rhs
- ` is used to build terms out of constructors, defined symbols and Java variables and functions

Exercise

- define a signature to handle sets
- define the operations
  - occurs : E x S -> Bool
  - union : S x S -> S
  - subset, eq : S x S -> Bool
  - diff : S x S -> S

Some implementation details

- How does Java know that it should print « suc(suc(zero())) » ?
- To each signature corresponds an implementation in Java

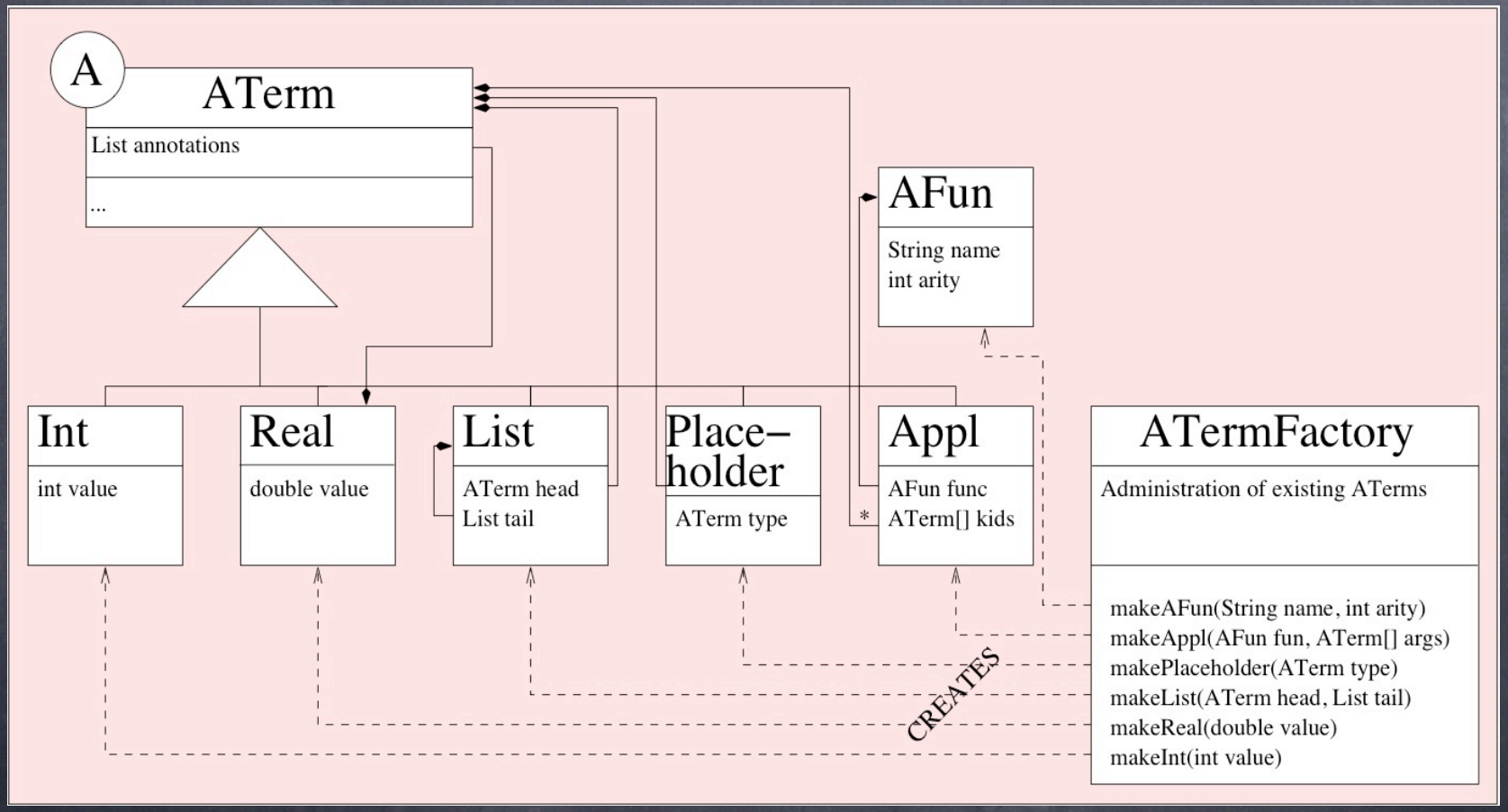
Gom

- Starting from a multi-sorted signature
- Generates the classes (based on ATerms) that allow us to represent the terms
  - maximal sharing
  - strong typing for terms
  - all typed term is an ATerm

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

ATerms


A
ATerm
List annotations
...
AFun
String name
int arity
Int
int value
Real
double value
List
ATerm head
List tail
Place–
holder
ATerm type
Appl
AFun func
ATerm[] kids
*
ATermFactory
Administration of existing ATerms
makeAFun(String name, int arity)
makeAppl(AFun fun, ATerm[] args)
makePlaceholder(ATerm type)
makeList(ATerm head, List tail)
makeReal(double value)
makeInt(int value)
CREATES

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

getAnnotation
setAnnotation
toString
etc.

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

getName
getArity

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

getAFun
getArgument
setArgument
etc.

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

getFirst
getNext
elementAt
insert
append
concat
etc.

ATerm library

- Aterm
  - AFun
  - ATermAppl
  - ATermList
  - ATermInt
  - ATermReal
  - ATermPlaceholder
  - ATermBlob
- ATermFactory

makeAFun
makeAppl
...
parse
readFromFile

etc.

Use ATerms

- Aterm t1 = factory.parse("a")
- Aterm t2 = factory.parse("f(a,b)")
- t2.getArgument(1) == t1 ?

  > true
- Aterm t3 = t2.setArgument(t1,2)

  > t3 = f(a,a)
- Aterm t4 = factory.parse("f(a,f(b))")

  > 'f' n'a pas de signature particulière
- ATerms are used to build terms, but there is no discipline imposed on their construction (signature, types)

Example of generated code

```
%gom {
    module Peano
    abstract syntax
    Nat  = zero()
         | suc(n:Nat)
}
```