Techniques de réécriture et transformations

Horatiu Cirstea
Pierre-Etienne Moreau

Informatique

- Science du traitement de l'information
  - Représentation des données
  - Transformation de ces données
- Réécriture
  - Abstraction où les données sont des termes
  - C'est un moyen d'étudier les relations qui existent entre les termes
  - C'est une façon de décrire des transformations de termes

Dans la suite

- Je vais présenter un langage fondé sur ces principes
  - Qui permet de représenter facilement des données arborescentes
  - Qui permet de les inspecter ou les transformer
- Intérêt par rapport à un langage classique
  - Plus haut niveau d'abstraction
  - On fait moins d'erreurs
  - On programme plus vite

Un exemple

+33* → France

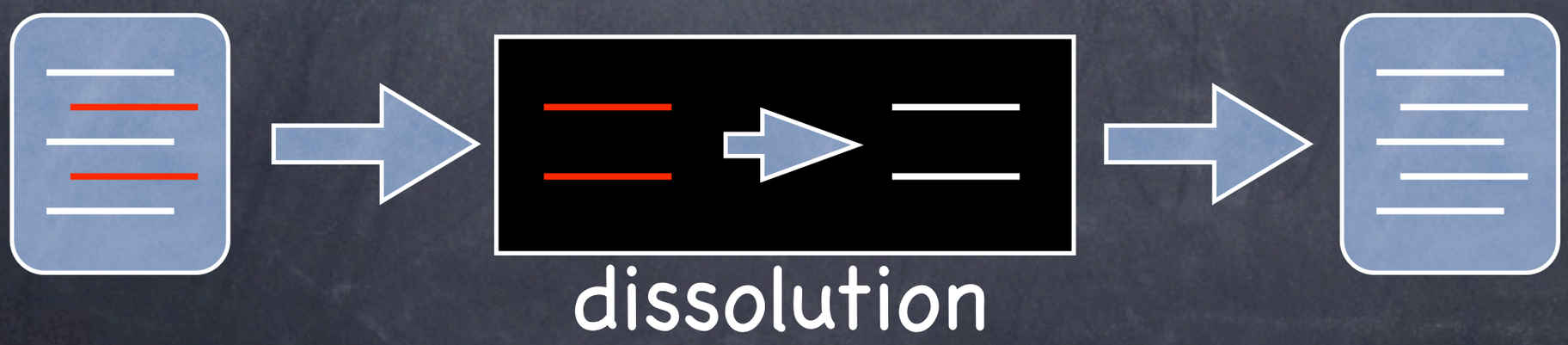
+333* → Est

+33383* → D.54

+0800* → Gratuit

Impératifs techniques

- Bonne interaction avec C, Java, ...
- Faciliter la communication avec l'extérieur (I/O, Corba, ...)
- S'adapter aux structures de données de l'utilisateur



Mise en pratique des îlots formels

- Un langage qui permet de :
  - Construire des structures arborescentes
  - Reconnaître des motifs dans une structure de donnée
- Lorsqu'un motif est reconnu
  - On dit que le motif « filtre » vers la structure
  - On peut exécuter une action, écrite en C ou Java

Présentation de Tom

Structure principale
les termes

- On se donne une signature many-sortée
- exemple : les entiers de Peano
  - zero : -> Nat
  - suc : Nat -> Nat

Gom permet de définir une signature

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

Règles de réécriture

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

Utilisation

- System.out.println(one())

> suc(zero())

- System.out.println(two())

> suc(suc(zero()))

Addition par réécriture

```
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()))

Exercice

- programmer la suite de Fibonacci
- fib(0) = 1
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2) pour $n \geq 2$

Système de réécriture

```
module Peano:rules() {
        one() -> suc(zero())
        two() -> suc(one())
        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)))
}
```

- le type des variables est inféré

Question

- peut-on écrire le programme suivant ?

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

Exercice

- Evaluer le terme 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()))
```

Question

- Que donne l'évaluation du programme suivant ?
- System.out.println(« res = » + fib(plus(one(),two())));

> res = suc(suc(suc(zero)))

- Que donne l'évaluation du programme suivant ?
- System.out.println(« res = » + suc(fib(plus(one(),two()))));
- ne compile pas : suc n'est pas une fonction Java

Mécanisme de base

- `suc(suc(zero()))
- permet de construire un terme

Le terme doit être bien formé et bien typé

Utilisation de `

- System.out.println(« res = » + `suc(fib(plus(one(),two()))));

```
> res = suc(suc(suc(suc(zero()))))
```

- Un ` peut contenir des constructeurs, des symboles définis
- À partir de la signature %gom, on peut représenter et afficher des termes

Utilisation en Java

- Nat one = `suc(zero());
- Nat two = `suc(suc(zero()));
- System.out.println(« one = » + one);
- System.out.println(« two = » + two);

> one = suc(zero)

> two = suc(suc(zero))

Question

- Peut-on écrire :
- Nat one = `suc(zero());
- Nat two = `suc(one);
- Oui
- Un ` peut contenir des constructeurs, des symboles définis et des variables Java

Résumé

- %gom permet de définir une signature
- rules() permet de définir un système de réécriture contenant :
  - des constructeurs et des variables dans le lhs
  - des constructeurs, des variables et des symboles définis dans le rhs
- ` permet de construire un terme à partir de constructeurs, symboles définis, variables et fonctions Java

Exercice

- définir une signature pour manipuler des ensembles
- définir les opérations
  - occurs : E x S -> Bool
  - union
  - subset, eq
  - diff

Question

- Comment Java sait qu'il doit afficher « suc(suc(zero())) » ?
- A chaque signature correspond une implantation en Java

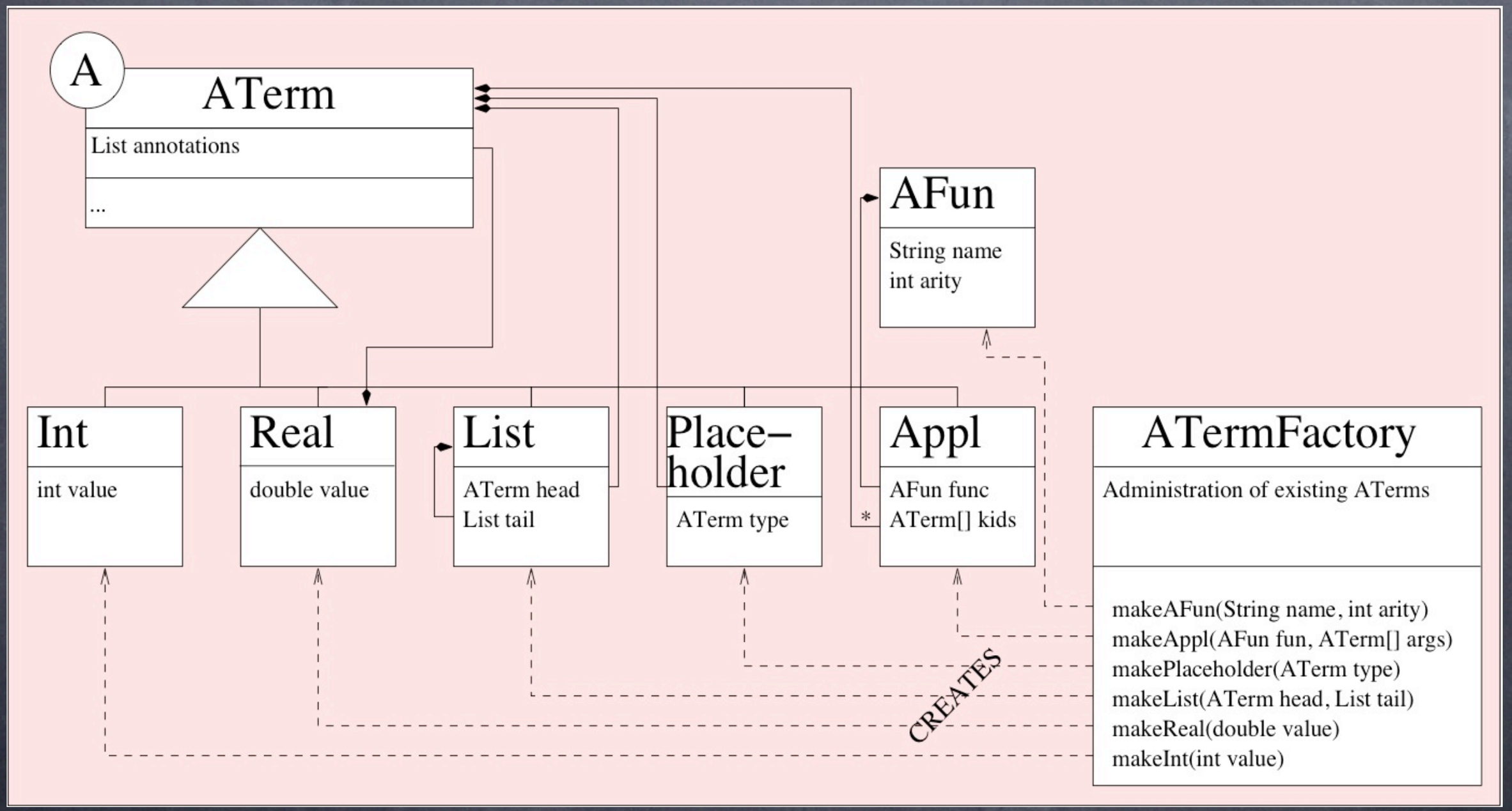
Gom

- A partir d'une signature multi-sortées
- Génère des classes, reposant sur les ATerms, permettant de représenter les termes
  - partage maximal
  - typage fort des termes
  - tout terme typé est un ATerm

Bibliothèque ATerm

- 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

Bibliothèque ATerm

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

getAnnotation
setAnnotation
toString
etc.

Bibliothèque ATerm

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

getName
getArity

Bibliothèque ATerm

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

getAFun
getArgument
setArgument
etc.

Bibliothèque ATerm

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

getFirst
getNext
elementAt
insert
append
concat
etc.

Bibliothèque ATerm

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

makeAFun
makeAppl
...
parse
readFromFile
etc.

Utilisation des 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
- Les ATerms permettent de construire des termes, mais il n'y a pas de sécurité (signature, types)

Exemple de code généré

```
%gom {
    module Expression
    imports int
    abstract syntax
    Expr = Cst(value:int)
         | Plus(e1:Expr, e2:Expr)
         | Mult(e1:Expr, e2:Expr)
    Bool = True()
         | False()
         | Equal(b1:Bool,b2:Bool)
}
```

Tom, la suite...

Exercice

- Définir une fonction

  plusInt(Nat, Nat) -> int

- Qui retourne l'addition de deux Nat sous forme d'entier

- System.out.println(« res = » + `plusInt(one,two));

- `> res = 3`

Solution (qui ne marche pas)

```
%gom {
    ...
    imports int
    ...
    int = plusInt(Nat, Nat)
    ...
}
module ...:rules() { ... }
```

Autre solution combiner filtrage et Java

```
public Nat plus(Nat n1, Nat n2) {
    %match(n1, n2) {
        x, zero() -> { return `x; }
        x, suc(y) -> { return `suc(plus(x,y)); }
    }
}
```

- plus n'est plus un symbole défini, mais une fonction Java
- x et y sont des variables instanciées par filtrage
- elles deviennent des variables Java locales à chaque règle

Exercice

- Ecrire le programme qui affiche le résultat sous forme d'entier
- System.out.println(« res = » + `entier(suc(plus(one,two))));
- `> res = 4`

Solution

```
public int entier(Nat n) {
    %match(n) {
      zero()    -> { return 0; }
      suc(x)    -> { return 1 + entier(`x); }
    }
}
```

Le membre droit peut mélanger Tom et Java

Sémantique de %match

- Sélectionne le premier pattern qui filtre
- Evalue l'action associée
- L'exécution reprend
  - après l'action (i.e. sélectionne le pattern suivant) s'il n'y a pas eu de break ou de return
  - en fonction du flot de contrôle

Question

- Peut-on encoder un système de réécriture conditionnelle avec le %match ?
- Peut-on encoder des motifs non-linéaires ?

Réponse

- Oui, il suffit de mettre une condition dans la partie action

```
public int entier(Nat n) {
    %match(n) {
        zero() -> { return 0; }
        suc(x) -> {
          if(`x != `zero()) { return 1 + `entier(x); }
        }
        suc(zero())  -> { return 1; }
    }
}
```