Implementing a Rule Based Language

Pierre-Etienne Moreau

July, 6th, 2007

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA

What would be a rewriting based language?

- rule: lhs ⇒ rhs
- conditional rule: lhs ⇒ rhs if cond
- given a term, computes a normal form

- syntax of a term:
  - prefix notation (a la ML or Lisp): eq(x,x) ⇒ true
  - user defined notation: x == x ⇒ true (more complex)

Many tools are based on rewriting

- CiME,
- daTac,
- Larch Prover,
- Otter,
- ReDuX,
- Reve,
- RRL,
- Spike,

- Caml,
- Clean,
- SML,
- ...

Some programming environments

main paradigm: term + rule + strategies

1975 Equational Interpreter (Chicago)

1977 OBJ (Menlo Park)

(Menlo Park) Maude

(Ishikawa) CafeOBJ

1985 ELAN (Nancy)

1980 ASF+SDF (Amsterdam)

1999 Stratego (Utrecht)

2001 Tom (Nancy)

ASF+SDF

Developed at CWI (Amsterdam)

Team headed by Paul Klint

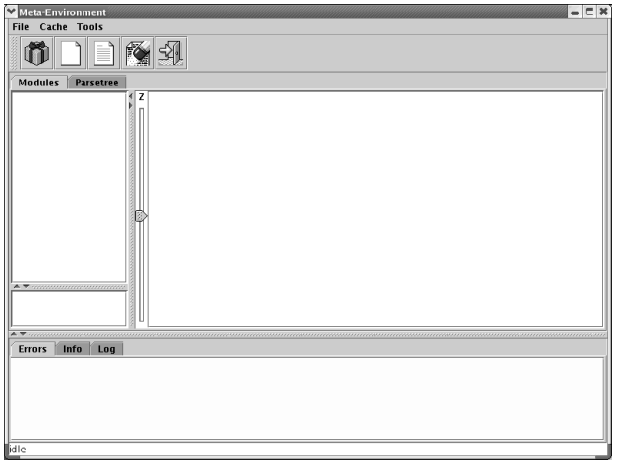
Goal:

- Develop specialized languages
- Analyse existing code (Cobol)
- Applications in banking domain

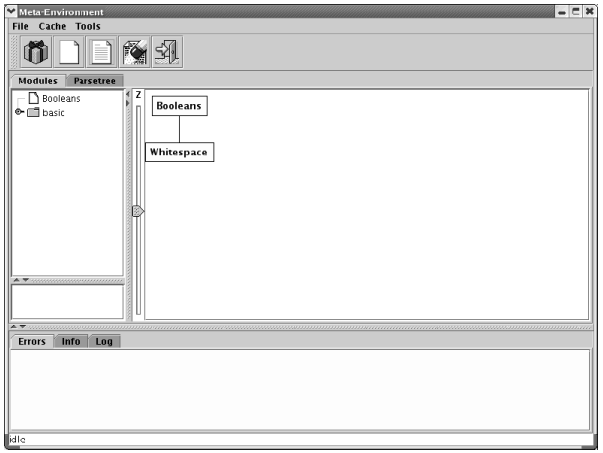
Consequences :

- graphical environment
- powerful syntax

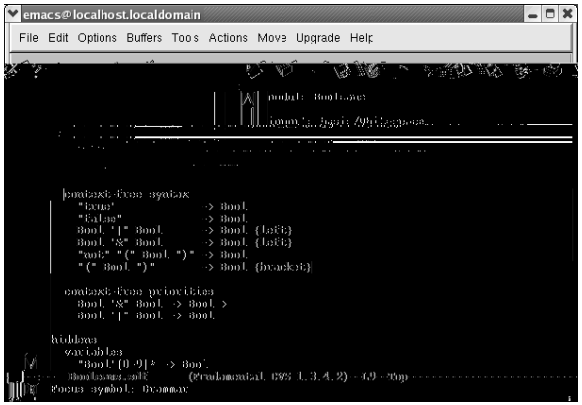
Running the environment



Defining a module



Editing the syntax



SDF

Powerful Syntax Definition Formalism

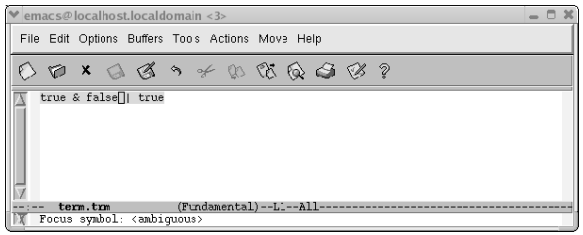
- many constructions to solve ambiguities

```
module Booleans
imports basic/Whitespace
exports sorts Bool
context-free syntax
  "true"             -> Bool
  "false"            -> Bool
  Bool "|" Bool      -> Bool {left}
  Bool "&" Bool      -> Bool {left}
  "not" "(" Bool ")" -> Bool
  "(" Bool ")"       -> Bool {bracket}
```

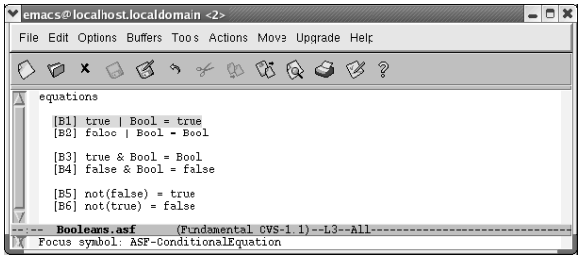

Solving ambiguities

```
context-free priorities
  Bool "&" Bool -> Bool >
  Bool "|" Bool -> Bool
Hiddens variables
  "Bool"[0-9]* -> Bool
```


Editing a term



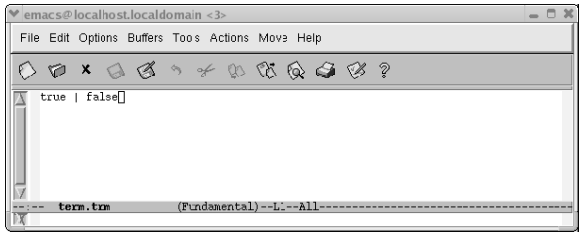
Editing rewriting rules

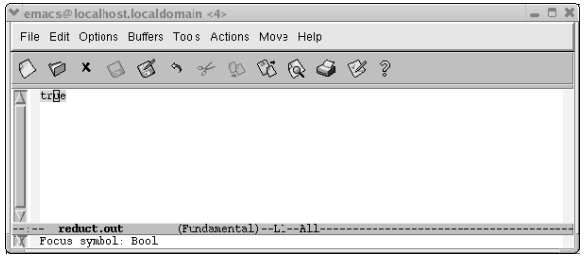


Equations

```
equations
[B1] true | Bool = true
[B2] false | Bool = Bool
[B3] true & Bool = Bool
[B4] false & Bool = false
[B5] not(false) = true
[B6] not(true) = false
```


Computing a normal form





OBJ, CafeOBJ, ELAN, Maude

Developed at SRI, JAIST, Loria, SRI

Teams headed by J. Goguen, K. Futatsugi, C. Kirchner, J. Meseguer

- make rewriting based languages usable in practice
- to develop automatic provers
- use the provers to prove properties about program written in the languages

Consequences:

- Equational theories (associativity, commutativity, …)
- Non terminating rewrite systems
- Strategies for controlling/exploring

ELAN
Main mechanism: rewriting

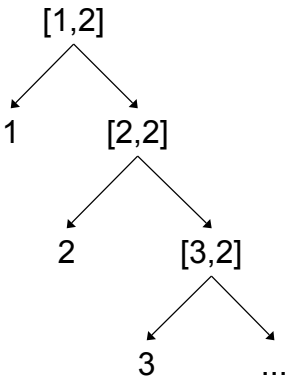
- unlabelled rule
- [] fact(1) ⇒ 1
- conditional rule
- [] fact(n) ⇒ n*fact(n-1) if n > 1

Computation

fact(3) ⇒ 3*fact(2) ⇒ 3*2*fact(1) ⇒
⇒ 3*2*1 ⇒ 3*2 ⇒ 6

Two problems

2 rules
- [] [x,y] → x
- [] [x,y] → [x+1,y]



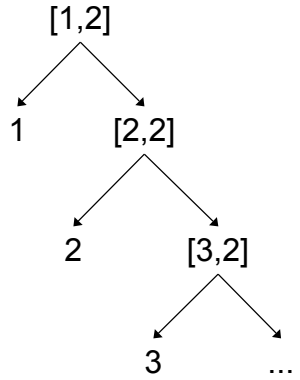
Non-confluent

Non-terminating

Controlling a non confluent system

Unnamed rule :
- [R1] [x,y] → x
- [R2] [x,y] → [x+1,y]

Strategy
- repeat(first one(R1,R2))
- repeat(dk(R1,R2))

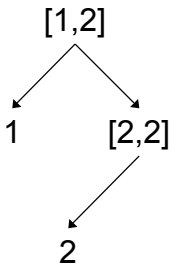


Controlling a non terminating system

Adding a condition :
- [R1] [x,y] → x
- [R2] [x,y] → [x+1,y] if x < y

Adding a rule :
- [R1] [x,y] → x
- [R2] [x,y] → [x+1,y]
- [check] [x,y] → [x,y] if x < y

Strategy
- repeat(dk(R1, check ; R2))



Maude Signature

```
fmod PEANO-NAT-EXTRA is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor iter] .
    op _+_ : Nat Nat -> Nat .
    vars M N : Nat .
    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```


Environment

```
Maude> reduce s(0) + s(s(0))
result Nat: s(s(s(0)))
```


Associative theory

```
fmod LIST is
    sorts Elt List .
    subsort Elt < List .
    op nil : -> List [ctor] .
    op _ _ : List List -> List [ctor assoc id: nil] .
endfm
```


Meta-level

```
op op_:_->_[_] :   Qid QidList Qid AttrSet -> OpDecl [ctor] .
```

meta-meta-encoding: of “ops t : -> Truth”

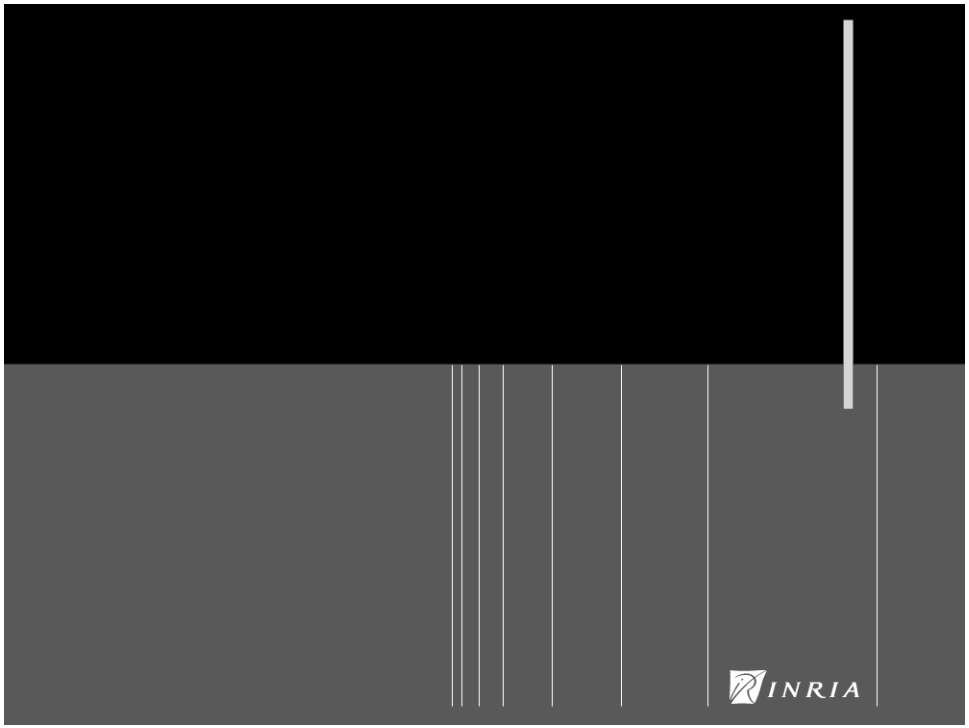
```
'__['op_:_->_`[_`].[ {''t}'Qid, {'nil}'QidList, {''Truth}'Qid, {'none}'AttrSet]
```

```
op fmod_is_sorts_.____endfm :
                    Qid ImportList SortSet SubsortDeclSet OpDeclSet
                    MembAxSet EquationSet -> FModule [ctor] .
```

Programming the meta-level

```
fmod 'PEANO-NAT-EXTRA is
    nil sorts 'Nat . none
    op '0 : nil -> 'Nat [ctor] .
    op 's : nil -> 'Nat [ctor] .
    op '_+_ : 'Nat 'Nat -> 'Nat [none] .
    None
    eq '_+_['N:Nat, '0.Nat] =    'N:Nat [none] .
    eq '_+_['s['M:Nat], 'N:Nat] =  's['_+_['M:Nat, 'N:Nat]]    [none] .
endfm

Maude> red metaReduce(
    ['PEANO-NAT-EXTRA],  '_+_['s['s['0.Nat]], '0.Nat]
) .
result ResultPair: {'s['s['0.Nat]], 'Nat}
```

Main principles

A rule is a pair

III → U

A subject

UMIIIMU

Rewriting: Searching and Replacing

A rule is a pair

III → U

A subject

UMIIIMU

Generalized to a triple

III → U if condition

Rewriting: Searching and Replacing

A rule is a pair

III → U

A subject

UMIIIMU

↓

UMUMU

Generalized to a triple

III → U if condition

Main problems

Perform a rewrite step

- Given a term and a rule, decide if the rule can can be applied
- We need a pattern matching algorithm
- Apply the rule
- We need to build a new term

Computing a normal form (wrt. a rewriting system)

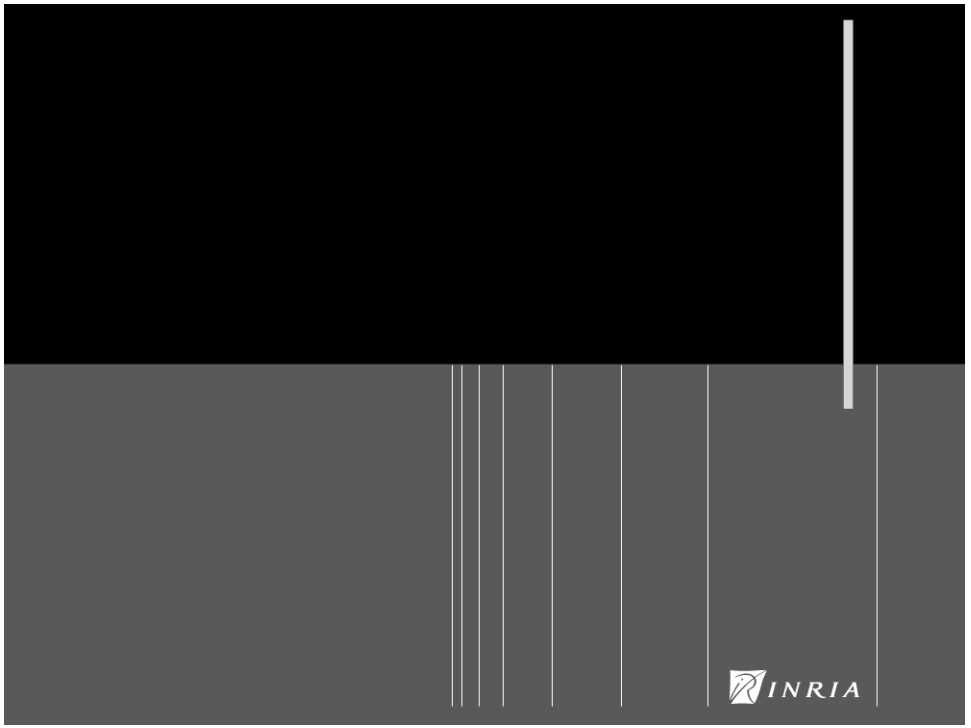
- Given a term and a rewrite system,
- Find a redex such that a rule can be applied
- Repeat until a fix-point is reached
- We need a strategy and a many-to-one matching algorithm

Perform pattern matching is « easy »

Decompose:	$(f\ P_1 \ldots P_n) << (f\ A_1 \ldots A_n)$	$\rightarrow \wedge_{i=1\ldots n}\ P_i << A_i$
SymbolClash:	$(f\ P_1 \ldots P_n) << (g\ A_1 \ldots A_n)$	→ False
Delete:	P << P	→ True
PropagageClash:	S ∧ False	→ False
PropagateSuccess:	S ∧ True	→ S

f(x,g(y)) << f(a,g(b))
x << a ∧ g(y) << g(b)
x << a ∧ y << b

f(b,g(y)) << f(a,g(b))
b << a ∧ g(y) << g(b)
False ∧ y << b
False



What does « compilation » mean?

My understanding:

- transform a program into another one (that is simpler to execute)

In other words:

- Solving a problem by executing instructions instead of manipulating data (interpretation)

Goals:

- Be efficient by generating specific code (better than a generic one)
- Reduce the number of dynamic memory allocations

Compiling pattern matching is simple

genMatching(termList, path, action) = match termList with:

- [appl@Appl(name,subterms),tail] → IfThen(cond,automata)
  cond = EqualFunctionSymbol(source,appl)
  source = Variable(PositionName(path))
  automata = genMatching(subterms, path, subAction)
  subAction = genMatching(tail, path, action)

Example: f(a,g(x))

- [Appl(« f », [
  - Appl(« a », [])
  - Appl(« g », [Variable(« x »)])])]
- if symb(s_0) = « f » then
  - …

Compiling pattern matching is simple (cont.)

genMatching(termList, path, action) = match termList with:

- [] → action
- [var@Variable(…),tail) → Let(var, source, subAction)
  source = Variable(PositionName(path))
  subAction = genMatching(tail, path', action)
- [Appl[...],tail] → ...

Example: f(a,g(x))

- [Appl(« f », [
  - Appl(« a », [])
  - Appl(« g », [Variable(« x »)])])]
- if symb(s_0) = « f » then
  - if symb(s_1) = « a » then
    - if symb(s_2) = « g » then
      - x := s_2_1

Compilation of two rules

f(a,g(x)) → ...

f(b,g(x)) → ...

- if symb(s_0) = « f » then
  - if symb(s_1) = « a » then …
    - if symb(s_2) = « g » then …
- if symb(s_0) = « f » then
  - if symb(s_1) = « b » then …
    - if symb(s_2) = « g » then …

The root symbol s_0 is tested several times

Other approach: many-to-one matching

Perform matching and select a rule at the same time

- static analysis of the rewrite system
- computation of a matching automaton
- generation of program (C, Java, assembly, …)

A lot of work:

- Hoffmann et O'Donnell (1982), Cardelli (1984), Peyton-Jones (1987), Gräf (1991), Sekar et al. (1992), Graf (1996), Nedjah et al. (1997), Moreau (1999), Maranget et al. (2001)

Non-deterministic matching automaton

3 rules

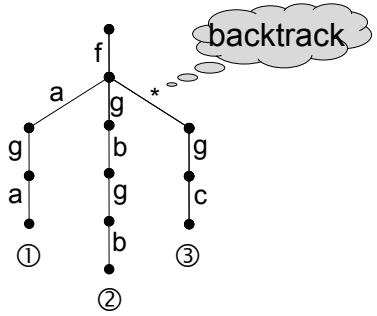
① f(a,g(a)) → a

② f(g(b),g(b)) → c

③ f(x,g(c)) → b

Non-determinism:
• to select a pattern
• to find all possible matches

Subject: f(g(a),g(c))



Deterministic matching automaton

3 rules

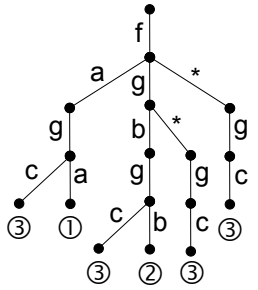
① f(a,g(a)) → a

② f(g(b),g(b)) → c

③ f(x,g(c)) → b

Gräf (1991)
Nedjah (1997)

Subject: f(g(a),g(c))



Deterministic matching automaton with jumpNode

3 rules

① f(a,g(a)) → a

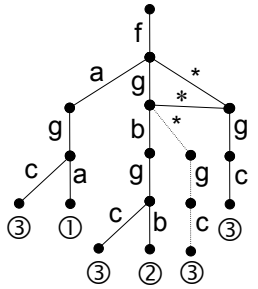
② f(g(b),g(b)) → c

③ f(x,g(c)) → b

• incremental construction

• reduced size

Subject: f(g(a),g(c))



Many-to-one matching

Given a term t

Given a rewrite system S

Compute the set of rules R of S such that lhs(R) matches t

This set can be represented by a vector of bits

Note: a singleton may be sufficient when there is no condition

Still to do:

- build the rhs (easy: allocate memory)
- find redex (easy: traverse the subject t)

A naïve algorithm

Select a position

Find a rule

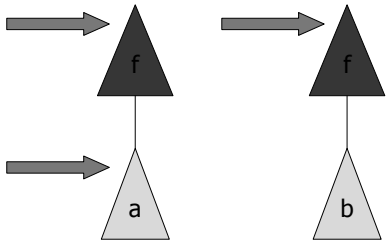
A couple (rule, position) may be tried several times

Example :

Term: f(a)

Rules:

- a → b ✖ ✔ ✖
- f(b) → c ✖ ✖ ✔



A possible optimization choose a reduction strategy

Example: leftmost-innermost

Drawbacks

- The reduction is not optimal
- The strategy can be non terminating

Advantages

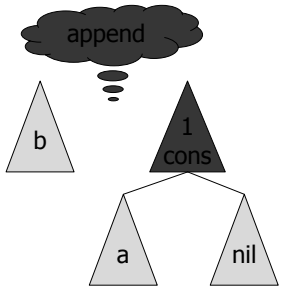
- Subterms are always in normal form when a rule is tried
- The normalization is performed during the construction of a term

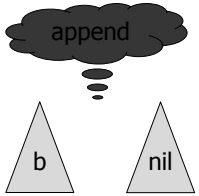
Example

append(e,nil) → cons(e,nil)

append(e,cons(h,t)) → cons(h,append(e,t))

Subject = append(b,cons(a,nil))



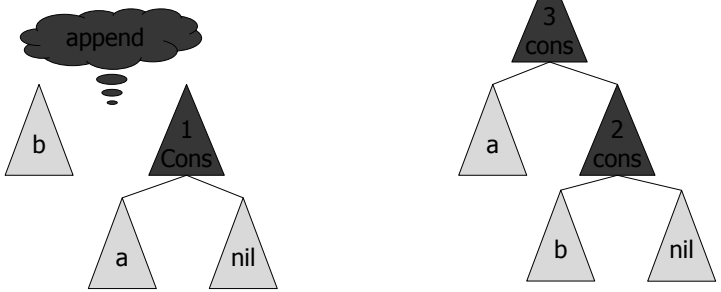


Example

append(e,nil) → cons(e,nil)

append(e,cons(h,t)) → cons(h,append(e,t))

Subject = append(b,cons(a,nil))



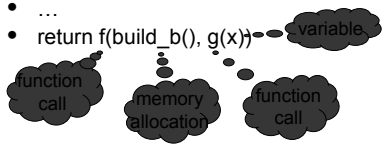
Construction of the rhs

To build a constructor
- memory allocation

To build a defined symbol
- function call, which selects a rule to fire

To build a variable
- the substitution is stored using low-level variables

Example: f(a,g(x)) → f(b,g(x))
- if symb(s) = « f » then
  - if symb(s_1) = « a » then
  - if symb(s_2) = « g » then x := s_2_1
- …
- return f(build_b(), g(x))



Key point of a good implementation

Reduce the number of allocations

Have a good memory allocator
- allocation in constant time
- automatic garbage collection

Two possible strategies
- mark and sweep
  - mark all living terms (size of living objects)
  - remove unmarked objects (size of the heap)
  - allocation using a list of free cells
- copy collector
  - move living object in another semi-space (size of living objects)
  - allocation in an array (constant time)

Improvement
- most of created symbols will be garbaged very soon
- generational copy collector (divide the heap into young and old objects)

Another strategy
Do not allocate a same object twice

Aterm: a standard term data-structure
- standardized input/output (similarly to XML)
- multi-platforms (C and Java)
- provides maximal sharing (using hash-code, i.e. hash-consing)
- provides a garbage collector (in C)
- efficient

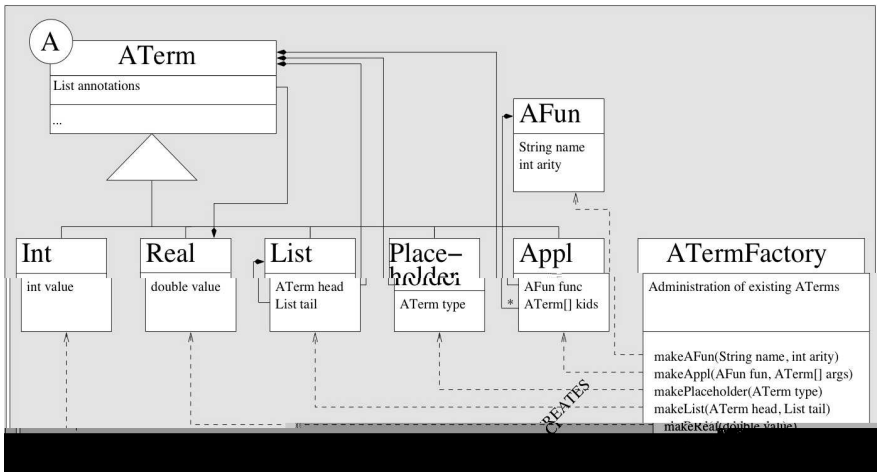
Aterm Library

Aterm

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

ATermFactory

ATerms



Aterm Library

Aterm

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

ATermFactory

getType

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.

Using the Aterm library in Java

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))")

Note: ‘f’ does not have a specific profile

Associative-commutative matching?

We have to work

Well known problem:

- Hullot (1980)
- Benanav et al. (1987)
- Kounalis et al. (1991)
- Bachmair et al. (1993)
- Lugiez et al. (1994)
- Eker (1995-2003)

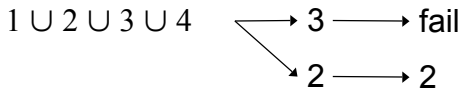
Associative-Commutative matching Source of non-determinism

Example: extract an even number

Rule: [extract] $x \cup E \rightarrow x$
[check] $x \rightarrow x$ if Even(x)

Strategy: dk(extract) ; dc one(check)

Subject: $1 \cup 2 \cup 3 \cup 4$



Problem to solve

Ordered canonical form:

$b+(a+(b+c)) = (b+a)+(b+c) = +(a,b^2,c)$

1 subject: +(f(a,a), f(a,g(b)), f(g(c),g(b)), g(a))

2 rules:

- +(z, f(a,x), g(a)) $\rightarrow r_1$
- +(f(a,x), f(y,g(b))) $\rightarrow r_2$

Problem to solve

Ordered canonical form:

$b+(a+(b+c)) = (b+a)+(b+c) = +(a,b^2,c)$

1 subject: +(f(a,a), f(a,g(b)), f(g(c),g(b)), g(a))

2 rules:

- +(z, f(a,x), g(a)) $\rightarrow r_1$
- +(f(a,x), f(y,g(b))) $\rightarrow r_2$

Questions we should ask

What do we want to do?

- be always efficient
- be very efficient on a restricted class of patterns
- support complex patterns
- support simple patterns only
- be quite efficient
- …

AC pattern matching is difficult to implement

Example (one-to-one)

❶ selecting a rule: +(z, f(a,x), g(a)) → r1

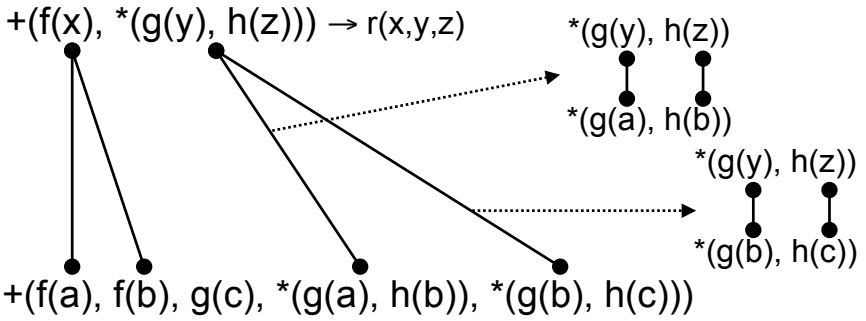
❷ building a Bipartite Graph:

Subject : +(f(a,a), f(a,g(b)), f(g(c),g(b)), g(a))

❸ solving a BG: x=a and z=+(f(a,g(b)), f(g(c),g(b)))

❹ computing another solution: x=g(b) and z=+(f(a,a), f(g(c),g(b)))

❺ selecting another rule: *everything has to be done again*



Building a BG is expansive

- recursive call to AC matching
- memory allocation

3 main ideas

Improve cases that occur in practice
- definition of a class of patterns

Reduce the BG construction cost
- definition of a compact BG data structure
- use syntactic matching automata

Integrate matching and normalization
- take external constraints into account (building substitution, rhs, …)

Classes of patterns

C_0 : linear terms
- example : a, x, f(x,y), …

C_1 : 1 level of AC symbol (semi-linear)
- 2 variables immediately under an AC symbol
- example : [+(z, y^3, f(a,x), g(a)) , *(g(t), b)]

C_2 : 2 levels of AC symbols
- example : *(x, +(y, z^2))

Sufficient in practice

Syntactic matching for subterms

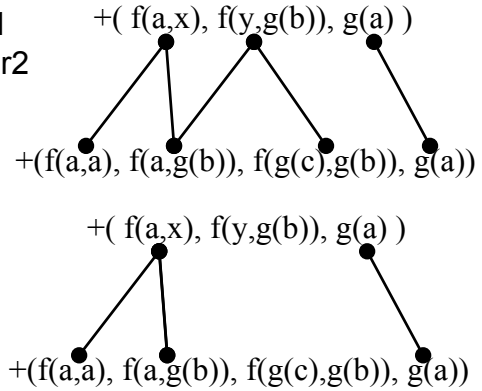
Many-to-one approach

❶ +(z, f(a,x), g(a)) → r1
+(f(a,x), f(y,g(b))) → r2

❷ building CBG with an automaton

❸ selecting a rule: extracting a BG

❹ computing a solution



Many-to-one approach

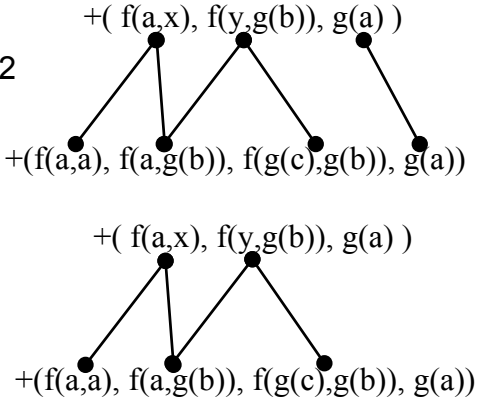
❶ +(z, f(a,x), g(a)) → r1
+(f(a,x), f(y,g(b))) → r2

❷ building CBG with an automaton

❸ selecting a rule: extracting a BG

❹ computing a solution

❺ selecting another rule : *extracting another BG*



Some key points

Data structure representation

- ordered canonical form
- flattened terms with multiplicities

Greedy algorithm

- compute at most one solution (sound, but not complete)
- efficient on a sub-class of patterns
- fast failure detection

Summary

- there exists several implementations of rule based languages
  - (ASF+SDF, ELAN, Maude, OBJ, Stratego, …)
- if you want to implement your own language
  - you need pattern matching (interpreter, compiler, one-to-one, many-to-one, deterministic, backtracking)
  - is the size of the generated code important ?
  - equational matching (A, AU, AC, ACU,…) ?
  - take care of the rhs construction
  - consider memory management problems
- next session
  - Interactive demo of the Tom language