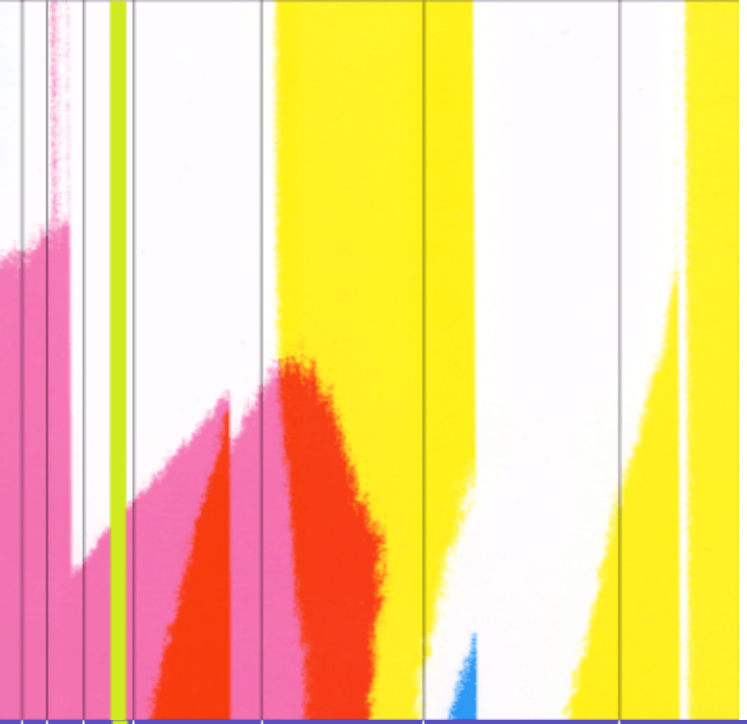


Implementing a Rule Based Language

Pierre-Etienne Moreau

Mines Nancy – Université de Lorraine

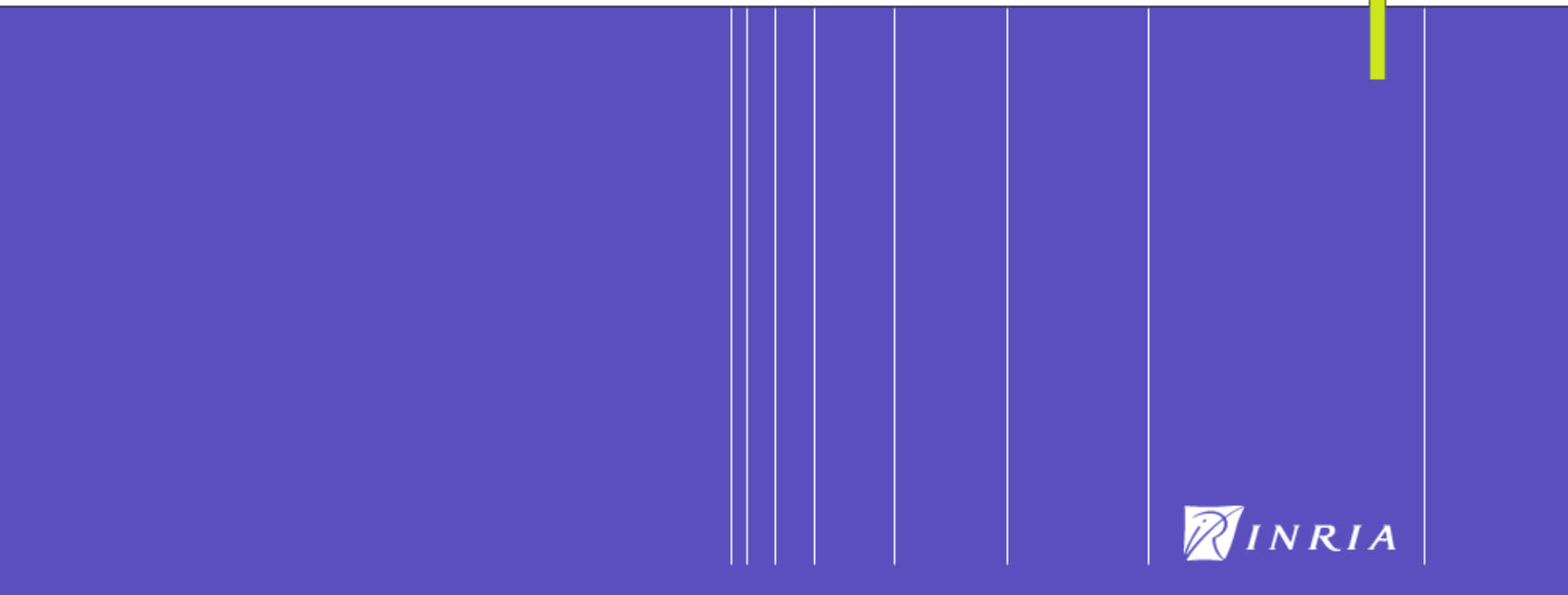
August, 26th, 2014

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Part I

Focus on Strategies



Strategies in Tom

A strategy can be applied to a term

The application results in a term or fails

Elementary constructors

Identity

Fail

Sequence

Choice

All

One

mu

Strategies in practice

A strategy is a first order object

- Strategy s = `Identity();

A strategy can be applied to a term

- Term result = s.visit(t);
- Either we get a result, either it fails (i.e. throws an exception)

A strategy is sort preserving

- MySort t = `a();
- MySort result = s.visit(t);

Elementary strategies

Rewrite rule

- A rule R : lhs → rhs is an elementary strategy
- Examples : R = a → b
- (R)[a] = b
- (R)[b] = fail
- (R)[f(a)] = fail

Identity and failure

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

Simple Demo

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

Exercise

Define:

- Try(s)
- Repeat(s)

User defined 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

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) or g(b,a)
- (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)))

Implementation

Demo

Defining a rule

```
%strategy Rule extends Fail() {
   visit Term {

       a() -> { return `b(); }

   }
}
```

- Rule is the name of the strategy
- Fail is the default behavior (I.e. the strategy fails when it cannot be applied)
- the strategy transforms nodes of sort Term

```
VisitableVisitor S = `Repeat(OnceBottomUp(Rule()))
Term result = S.visit(`f(a(),g(b(),a())));
```

Strategy library

Common strategies can be defined using parameters and recursion (mu)

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

VisitableVisitor OnceBottomUp(VisitableVisitor S) {
    return `mu(MuVar("x"),Choice(One(MuVar("x")),S)) }
}
```

Prototyping pattern matching

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

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

Part II

Implementation of rewriting

Main principles of a rule based language

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 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:	$(f\ P_1 \ldots P_n) << (g\ A_1 \ldots A_n)$	$\rightarrow$ False
Delete:	$P << P$	$\rightarrow$ True
PropagageClash:	$S \wedge$ False	$\rightarrow$ False
PropagateSuccess:	$S \wedge$ True	$\rightarrow S$

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

Do we want to be efficient? Compilation?

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

Representing a pattern

Given a pattern: f(a,g(x))

We need a generic representation in the compiler:

- Term = Appl(String, List)
- List = Cons(Term,List) | Empty()

Example: f(a,g(x))

- [Appl(« f », [
  - Appl(« a », [])
  - Appl(« g », [Variable(« x »)])])]

Compiling pattern matching is simple

compilation(*patterns*, path, action) = match *patterns* with:

- [appl@Appl(name,subterms), others] → IfThen(cond,automata)
  - cond = EqualFunctionSymbol(appl, subject)
  - subject = Variable(PositionName(path))
  - automata = compilation(subterms, path, subAction)
  - subAction = compilation(others, path, action)

Example: *patterns* = [f(a,g(x))]

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

Compiling pattern matching is simple (cont.)

compilation(*patterns*, path, action) = match *patterns* with:

- [] → action
- [var@Variable(…),others) → Let(var, subject, subAction)
  subject= Variable(PositionName(path))
  subAction = compilation(others, path', action)
- [Appl[...],others] → ... /* see previous slide */

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

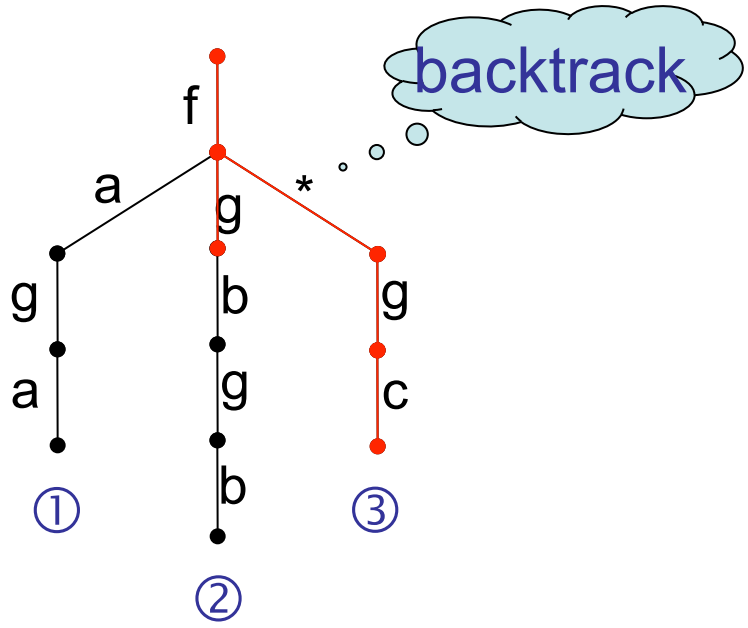
① f(a,g(a)) → a

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

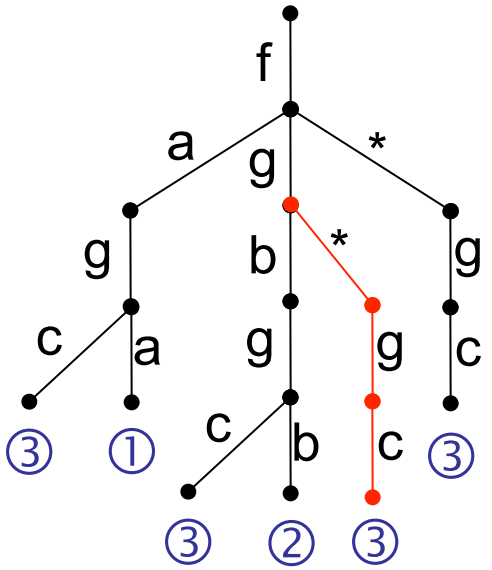
3 rules

① f(a,g(a)) → a

② f(g(b),g(b)) → c

③ f(x,g(c)) → b

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



Gräf (1991)
Nedjah (1997)

Deterministic matching automaton with jumpNode

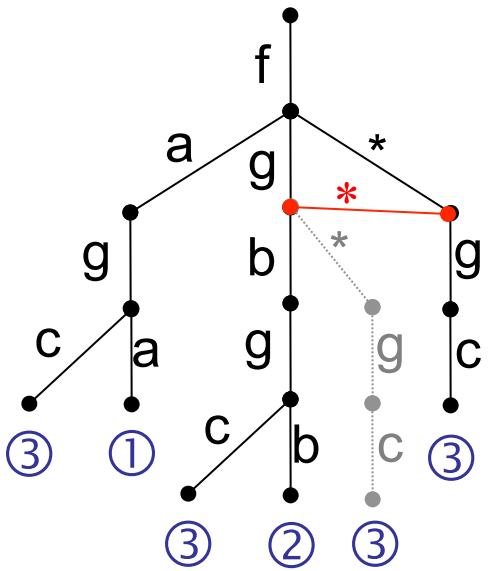
3 rules

① f(a,g(a)) → a

② f(g(b),g(b)) → c

③ f(x,g(c)) → b

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



- incremental construction
- reduced size

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

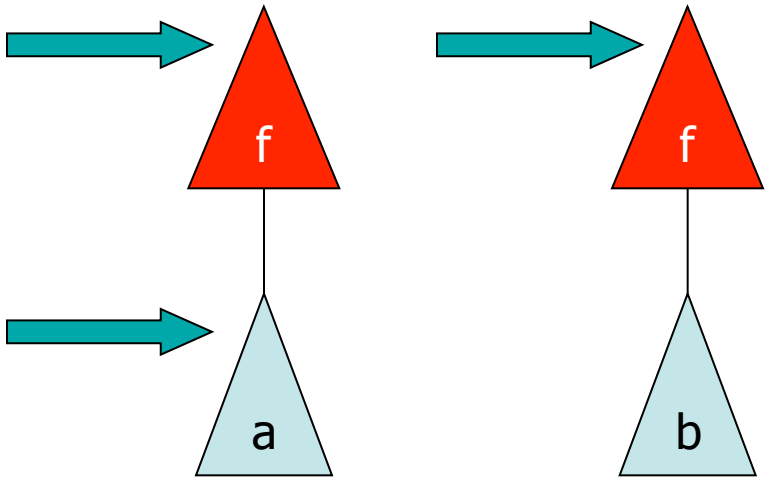
A couple (rule, position) may be tried several times

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

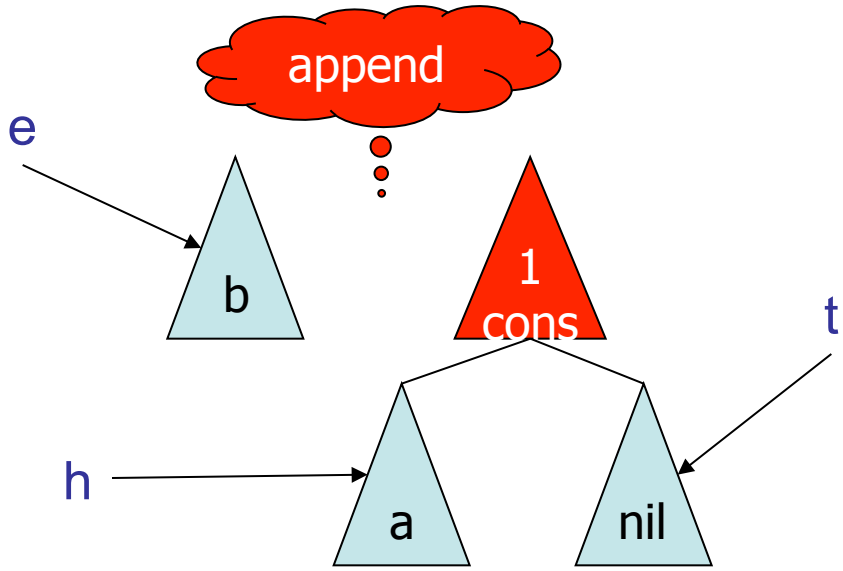
- Subterms are always in normal form when a rule is tried
- The normalization is performed during the construction of a term

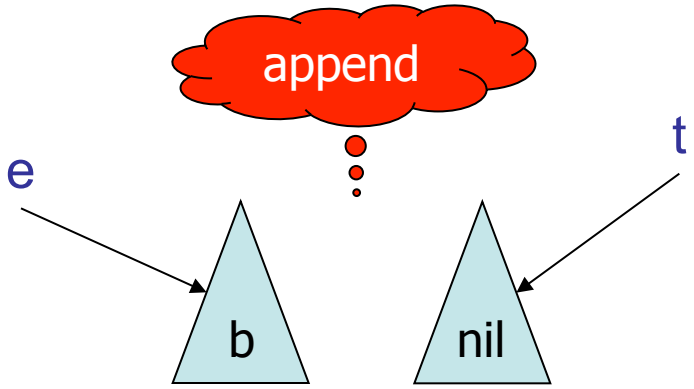
Example

append(e,nil) → cons(e,nil)

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

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



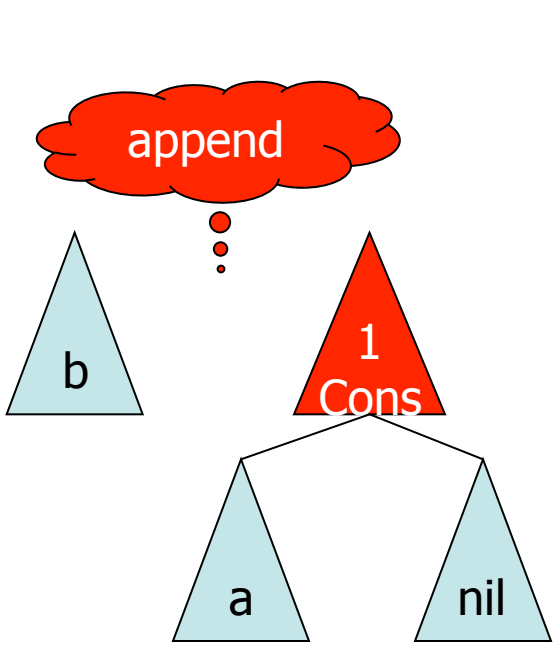


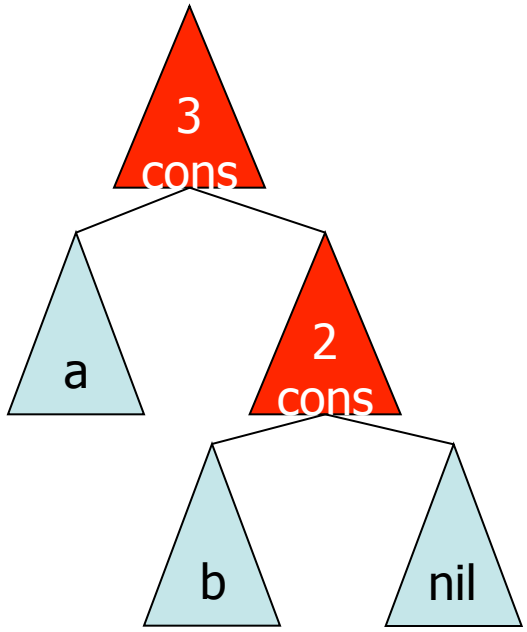
Example

append(e,nil) → cons(e,nil)

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

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





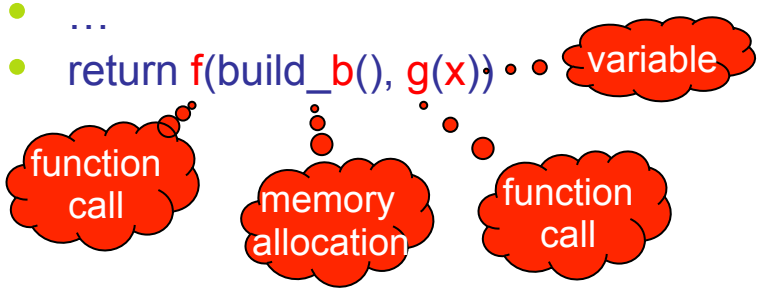
Construction of the rhs

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

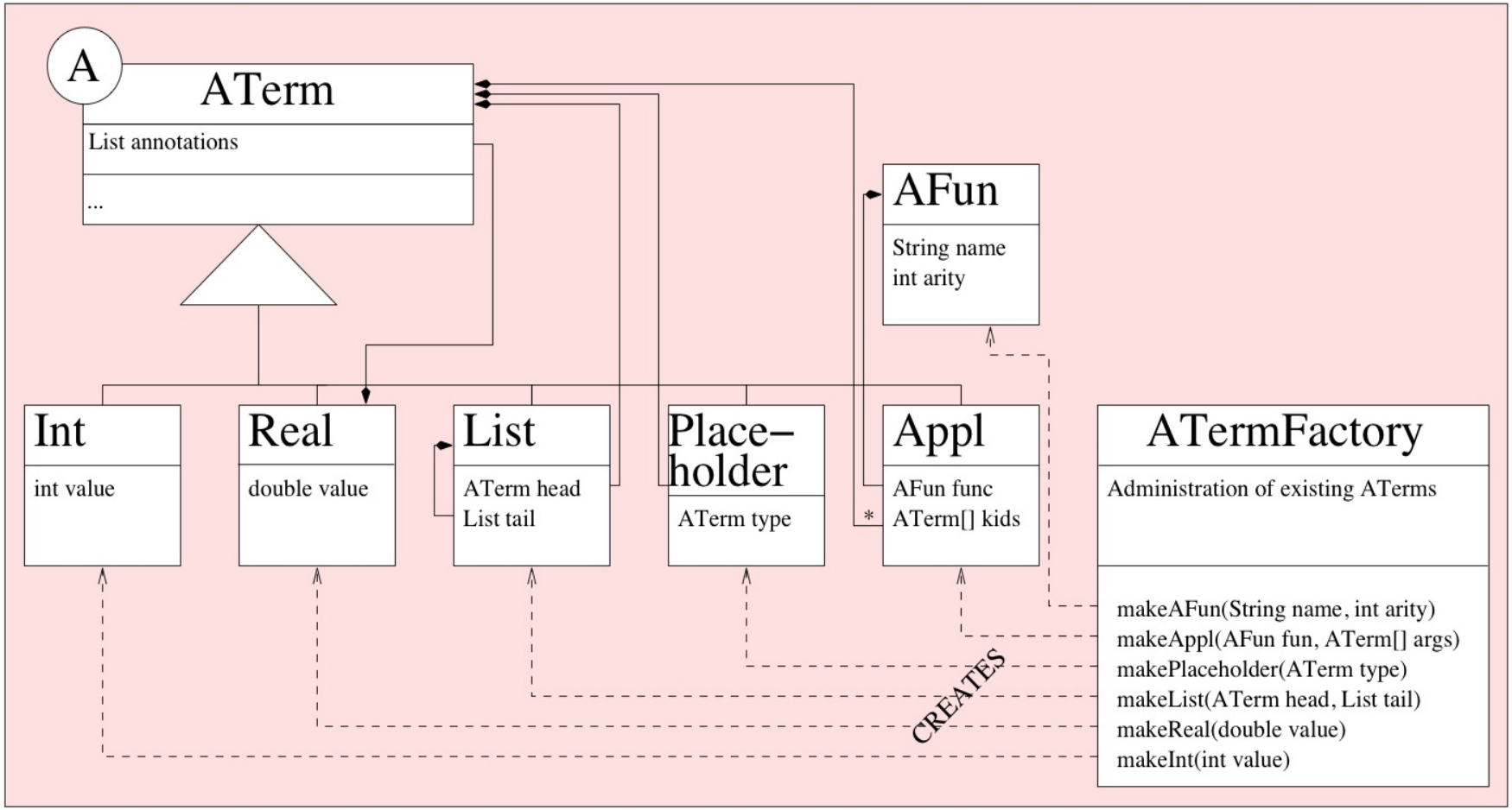
Aterm Library

Aterm

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

ATermFactory

ATerms


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

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

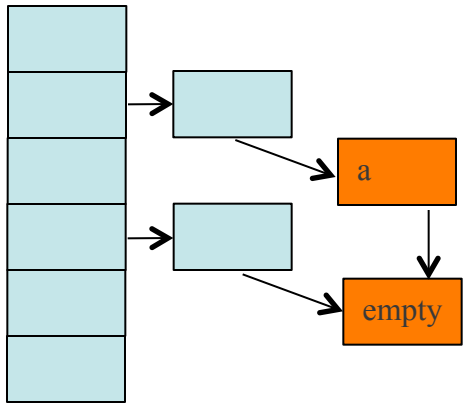
Implementing the Aterm library

Aterm t1 = factory.makeAppl("a", factory.makeList())

Aterm t2 = factory.makeAppl("b", factory.makeList())

Aterm t3 = factory.makeAppl("f", factory.makeList(t1, t2))

- Get hashcode of empty list
- Compute hashcode of "a"
- Compute hashcode of t1
- Allocate memory and store t1 in hashtable



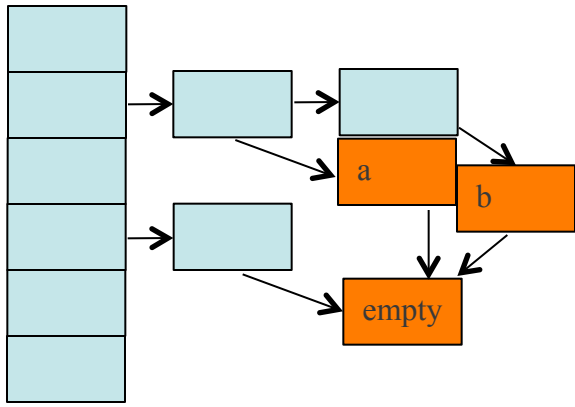
Implementing the Aterm library

Aterm t1 = factory.makeAppl("a", factory.makeList())

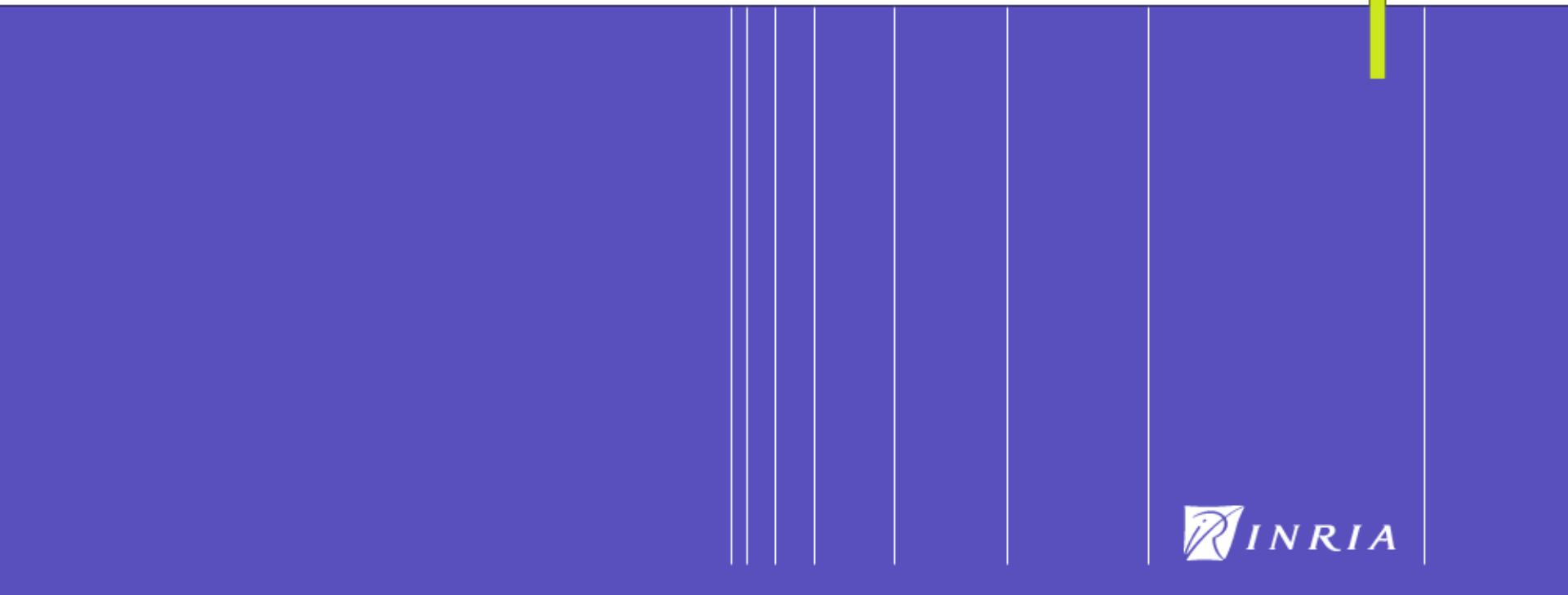
Aterm t2 = factory.makeAppl("b", factory.makeList())

Aterm t3 = factory.makeAppl("f", factory.makeList(t1, t2))

- Get hashcode of empty list
- Compute hashcode of "b"



Do we want more?



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)

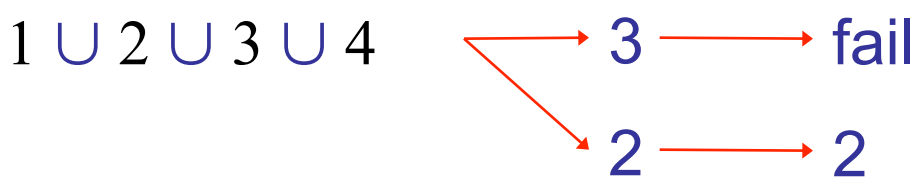
Associative-Commutative matching
Source of non-determinism

Example: extract an even number

Rule: [extract] x ∪ E → x
[check] x → x if Even(x)

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

Subject: 1 ∪ 2 ∪ 3 ∪ 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

More, more, and more

Strategies can be used to:

- perform non-deterministic computation (compute a set of results)
- cut branches in a search
- find a position in a term

Strategies are:

- quite easy to use
- difficult to implement, but easy to maintain (~1000 lines)
- efficient (may be 10% slower than an optimized ad-hoc implementation)
- reusable

The Tom compiler is written using strategies

Available at http://tom.loria.fr

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