# Compilation of AU-matching in Tom

Pierre-Etienne Moreau
LORIA - INRIA Lorraine

# Possible subtitles

- How to make rewriting usable in practice
- How to make our technology available
- How to share and reuse our software

# Motivations

- After 20 years of research in rewriting
- We are convinced that:
  - specifying by rewriting is a nice idea
- But, we also want to:
  - make rewriting usable in practice
  - integrate rewriting into existing programming environments

# Goal and approaches

- Introduce pattern matching facilities into languages like C or Java
  - Encapsulate a rewrite engine into a library
  - Provide a compiler for rewriting
- Questions to answer
  - Implement a matching algorithm again?
  - In which language? C? Java? Caml?
  - For which term data-structure? A fixed one?

# Main ideas

- Make matching algorithms
  - Simple
  - Independent on the data-structure
  - Independent on the host language
- How?
  - By designing generic algorithms
  - By introducing a mapping from concrete to algebraic data types
  - By generating kernel intermediate constructs

# Tom

- A pattern matching processor for C and Java which offers:
  - a signature definition formalism (%type and %op)
  - new term-based constructs (%rule, %match, and `)
- A concept presented at LDTA'2001 and CC'2003
- A stable implementation: The Tom compiler is written in Tom itself
- A system available at: tom.loria.fr

# The language

```
%type Term {
  implement { ATerm }
  get_fun_sym(t) { t.getAFun() }
  cmp_fun_sym(t1,t2) { t1 == t2 }
  get_subterm(t, n) { t.getArgument(n) }
  equals(t1, t2) { t1 == t2}
}

%op Term suc(Term) {
  fsym(t) { makeAFun("suc",1) }
}
```

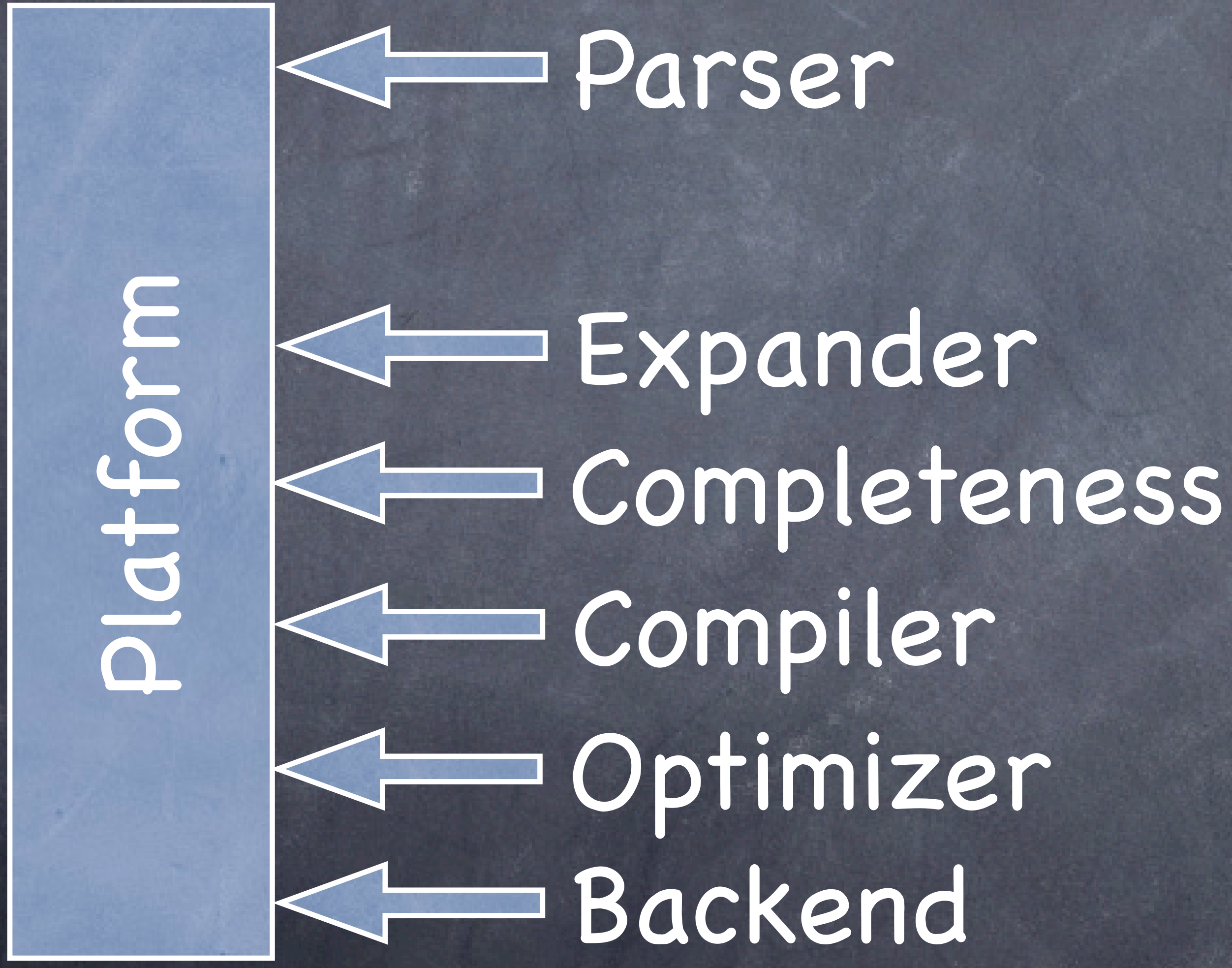
```
Term plus(Term t1, Term t2) {
  %match(t1,t2) {
    x, zero -> { return `zero; }
    x, suc(y) -> { return `suc(plus(x,y)); }
  }
}

List sort(List t) {
  %match(t) {
    conc(X*,a,Y*,b,Z*) -> {
      if greaterThan(a,b)
        return `sort(conc(X*,b,Y*,a,Z*)); }
    _ -> { return t; }
  }
}
```

# Technical considerations

- How to build an implementation for this language:
  - “easy” to maintain and extend (no global state)
  - support for research (without modifying a line)
  - allows collaborative development (use AST)

# Plugin based architecture



Platform

Parser

Expander

Completeness

Compiler

Optimizer

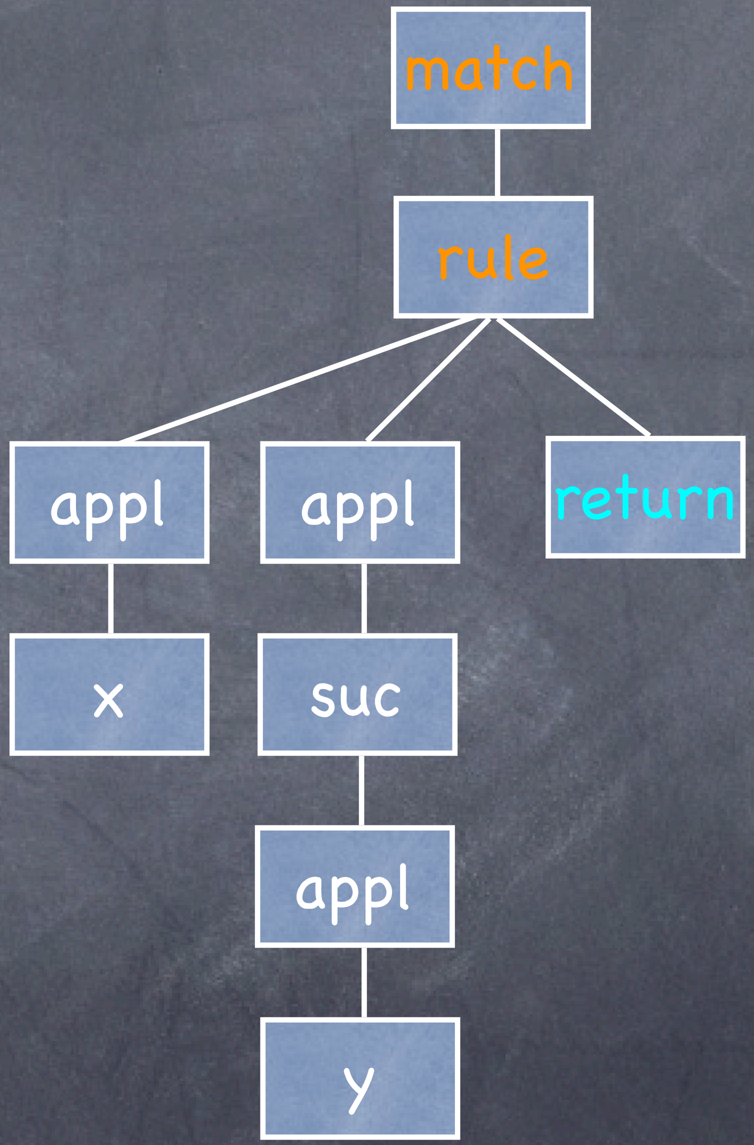
Backend

- Each phase is a plugin
- AST data structure
- Plugins can be added
- Without modifying the source of the compiler

# Parser

- Read input file
- Parse Tom constructs
- Build AST

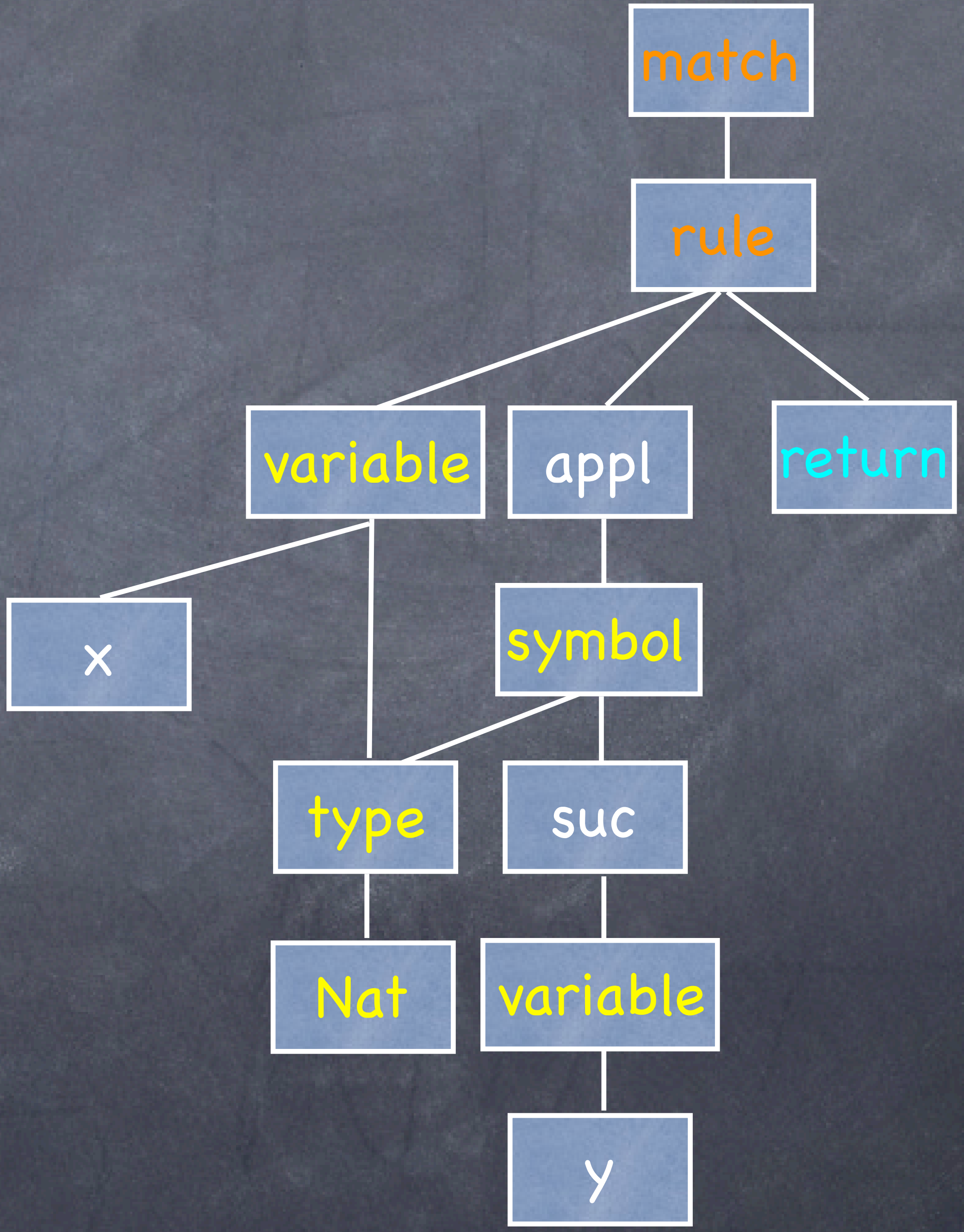
```
print(s1)
%match(s1,s2) {
  x,zero -> return x
  x,suc(y) -> ...
}
```



# Expander

- Perform type inference
- Introduce variables

```
print(s1)
%match(s1,s2) {
  x,zero -> return x
  x,suc(y) -> ...
}
```



match
rule
variable
appl
return
x
symbol
type
suc
Nat
variable
y

# Compiler

- Takes an AST and generates target language
- Constraints:
  - keep the software simple
  - handle multiple target languages
- Solution:
  - split the compiler in two parts
  - introduce an intermediate abstract language (C ∩ Java ∩ Caml)

# Pre-compiler

- Perform program transformation
  - rules are expanded into match+build
  - non-linear patterns are linearized
  - nested A-patterns are flattened (abstraction variables are introduced)
  - high level optimizations are performed

# Kernel-compiler

- Transform a simplified matching construct (flattened linearized patterns) into a discrimination tree (nested if-then-else and loop)
  - we use an abstract language to describe the matching automata
  - the subject-term is “seen” via destructors
    - the abstract language can be compiled by various back-ends (C, Java, Caml)

# Intermediate language

- Instruction ::=
  - Let(Term, Expression, Instruction)
  - IfThenElse(Expression, Instruction, Instruction)
  - WhileDo(Expression, Instruction)
  - Block(InstructionList)
  - Nop

- Expression ::=
  - Not(Expression)
  - IsEmpty(Term)
  - EqualFunctionSymbol(Term,Term)
  - GetSubterm(Term, Number)
  - GetHead(Term)
  - GetTail(Term)
- Term ::=
  - Variable(Name, Type)
  - ...

# Compilation of matching

f(a,g(x)) -> ...

- genFreeMatching(termList, path, action) = match termList with:
  - nil → action

    \+ IfThenElse
  - cons(var@Variable[...],tail) → Let(var, source, subAction)
    - where source = Variable[PositionName(path)]
    - subAction = genFreeMatching(tail, path’, action)
  - cons(Appl[...],tail) → ...

- genFreeMatching(termList, path, action) = match termList with:
  - cons(appl@Appl[args=subterms],tail) → IfThenElse(cond,automata,Nop)
    - cond = EqualFunctionSymbol(source,appl)
    - source = Variable[PositionName(path)]
    - automata = genFreeMatching(subterms, path, subAction)
    - subAction = genFreeMatching(tail, path', action)
  - cons(Appl[args=subterms,theory=list],tail) → genListMatching(termList, path, action)

# Associative operator

- we consider f, where $f(x,f(y,z)) = f(f(x,y),z)$
- f is AU when $f(x,e) = f(e,x) = x$
- solving $p \ll s$, consists in finding a set of substitutions $\Sigma$ such that $\forall\sigma \in \Sigma, p\sigma = s$
- we usually consider flattened forms: $f(x,y,z)$
- $f(x,y) \ll f(a,b,c)$ gives $\Sigma = \{ x=a, y=f(b,c) \} \cup \{ x=f(a,b), y=c \}$
- How to restrict x to an instance of “size=1”?

# Controlling matching

- We usually introduce (many/ordo) sorted signature and injection operators
  - f: L × L → L (associative operator)
  - g: E → L (injection operator)
- f(a,b,c) becomes f(g(a),g(b),g(c))
- f(g(x),y) << f(g(a),g(b),g(c)) gives $\Sigma = \{ x=a, y=f(g(b),g(c)) \}$

# Associative operator in Tom

- A, AU operators are difficult to use without invisible operator or ordo-sorted signature
- conc: E × ... × E → L (written E* → L)
- conc(a,b,c) is a valid term
- we use X* to denotes a variable of sort L
- conc(X*,Y*) corresponds to f(x,y)
- conc(x,Y*,z) corresponds to f(g(x),y,g(z))
- X* can be instantiated by the empty list

# AU-matching vs. Tom

| | |
|---|---|
| AU-operator: $L \times L \rightarrow L$ | List-operator: $E^* \rightarrow L$ |
| f(x,f(y,z))=f(f(x,y),z)<br>f(x,e)=f(e,x)=x | conc(x,conc(y,z))=conc(x,y,z)<br>conc(x,conc())=conc(x) |
| f(g(x),g(b),y) | conc(x,b,Y*) |

# Associative matching

- genListMatching(termList, path, action) = match termList with:
  - nil → IfThenElse(IsEmptyList(subject), action, Nop)
- conc() << t -> action is compiled into:

```
if IsEmpty(t) then
  action
endif
```

- cons(var@Variable[…],tail) → IfThenElse(Not(IsEmpty(subject)), Let(var, GetHead(subject), Let(subject, GetTail(subject), subAction))
  - where subAction = genListMatching(tail, path’, action)
- conc(x,…) << t -> action is compiled into:

```
if Not(IsEmpty(t)) then
   x = GetHead(t)
   t = GetTail(t)
   action
endif
```

# A more complex case

- cons(var@VariableStar[...],tail) →
- We distinguish 3 cases:
  1. tail = Nil: Let(var, subject, subAction)
  2. tail only contains VariableStar
  3. other
- Subcase 1: tail = Nil
  - conc(X*) << t -> action is compiled into: X = t

# Subcase 2: tail only contains VariableStar

- cons(var@VariableStar[...],tail) →
- conc(X*,Y*) << t -> action is compiled into:

```
list1 = t
begin1 = list1
end1 = list1
do
  X = GetSlice(begin1,end1)
  Y = list1
  action
  if Not(IsEmpty(end1)) then
    end1 = GetTail(end1)
  endif
  list1 = end1
while Not(IsEmpty(end1))
```

We build the AST
that corresponds to
this generated code

During the workshop, we have
discovered a fail in this
subcase: this shows that
certification is needed

# Subcase 3: general case

- cons(var@VariableStar[...],tail) →
- conc(X*,f(v),Y*) << t -> action is compiled into:

```
list1 = t
begin1=list1;
end1=list1;
while Not(IsEmpty(end1)) do
  list1 = end1
  X = GetSlice(begin1,end1);
  elt = GetHead(list1);
  list1 = GetTail(list1);
  sub-pattern-matching(elt,action)
  end1 = GetTail(end1);
endwhile
```

# Example of generated code

```
%match(List subject) {
  List(X1*,f(v),X2*) -> {
    System.out.println(`v);
  }
}
```

```
if(tom_is_fun_sym_List(subject)) {
  List list1 = subject;
  while(!(tom_is_empty_List(list1))) {
    Element elt = tom_get_head_List(list1);
    if(tom_is_fun_sym_f(elt)) {
      System.out.println(tom_get_slot_f_v(elt));
    }
    list1 = tom_get_tail_List(list1);
  }
}
```

# Element of conclusion

- By being data-structure and language independent
  - Tom is a bridge between the rewriting community and the OO community
  - It can be used to implement normalization procedures

# Recent and future developments

- Tom offers XML transformation facilities by translating XML patterns into AU-patterns
- Based on Elan/Stratego/JJTraveler, we have defined a library for traversal strategies
- Current projects:
  - Certify the generated code
  - Extend pattern matching to AC matching

Thanks