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

- 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) -> ...
}
```
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

\[
f(a, g(x)) \rightarrow \ldots
\]

\[
\text{genFreeMatching}(\text{termList}, \text{path}, \text{action}) = \\
m\text{atch } \text{termList with:}
\]

- \text{nil} \rightarrow \text{action} \quad + \text{IfThenElse}
- \text{cons}(\text{var@Variable}[...], \text{tail}) \rightarrow \text{Let}(\text{var}, \text{source, subAction})
  - \text{where source} = \text{Variable}[\text{PositionName}(\text{path})]
  - \text{subAction} = \text{genFreeMatching}(\text{tail, path', action})
- \text{cons}(\text{Appl}[...], \text{tail}) \rightarrow \ldots
genFreeMatching(termList, path, action) =
match termList with:
  
  cons(appl@Appl[wargs=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[wargs=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

- $f: L \times L \to L$ (associative operator)
- $g: E \to L$ (injection operator)

$f(a,b,c)$ becomes $f(g(a),g(b),g(c))$

$f(g(x),y) \ll f(g(a),g(b),g(c))$ 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: $E \times \ldots \times E \rightarrow L$ (written $E^* \rightarrow L$)

- 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

<table>
<thead>
<tr>
<th>AU-operator: $L \times L \rightarrow L$</th>
<th>List-operator: $E^* \rightarrow L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x, f(y, z)) = f(f(x, y), z)$</td>
<td>$\text{conc}(x, \text{conc}(y, z)) = \text{conc}(x, y, z)$</td>
</tr>
<tr>
<td>$f(x, e) = f(e, x) = x$</td>
<td>$\text{conc}(x, \text{conc}()) = \text{conc}(x)$</td>
</tr>
<tr>
<td>$f(g(x), g(b), y)$</td>
<td>$\text{conc}(x, b, Y^*)$</td>
</tr>
</tbody>
</table>
Associative matching

\[
\text{genListMatching}(\text{termList}, \text{path}, \text{action}) = \text{match termList with:}
\]

- \text{nil} \rightarrow \text{IfThenElse}(\text{IsEmptyList}(\text{subject}), \text{action}, \text{Nop})

\[
\text{conc()} \ll t \rightarrow \text{action} \quad \text{is compiled into:}
\]

\[
\text{if IsEmpty}(t) \text{ then}
\quad \text{action}
\quad \text{endif}
\]
cons(var@Variable[...], tail) →
IfThenElse(Not(IsEmpty(subject)), Let(var, GetHead(subject), Let(subject, GetTail(subject), subAction))

  where subAction = genListMatching(tail, path', action)

cconc(x,...) << t → action is compiled into:
if Not(IsEmpty(t)) then
  x = GetHead(t)
  t = GetTail(t)
  action
endif
A more complex case

\[ \text{cons}(\text{var}@\text{VariableStar}[...], \text{tail}) \rightarrow \]

We distinguish 3 cases:

1. \(\text{tail} = \text{Nil} \): \texttt{Let}(\text{var}, \text{subject}, \text{subAction})
2. \(\text{tail} \) only contains \text{VariableStar}
3. other

Subcase 1: \(\text{tail} = \text{Nil}\)

\[ \text{conc}(X^*) \ll t \rightarrow \text{action} \] is compiled into: \(X = t\)
Subcase 2: tail only contains VariableStar

\[ \text{cons}(	ext{var}@\text{VariableStar}[], \text{tail}) \rightarrow \]

\[ \text{conc}(X^*,Y^*) \ll t \rightarrow \text{action} \]

is compiled into:

\[
\text{list1} = t \\
\text{begin1} = \text{list1} \\
\text{end1} = \text{list1} \\
\text{do} \\
\quad \text{X} = \text{GetSlice}(\text{begin1}, \text{end1}) \\
\quad \text{Y} = \text{list1} \\
\quad \text{action} \\
\quad \text{if Not(IsEmpty(\text{end1})) then} \\
\quad \quad \text{end1} = \text{GetTail(\text{end1})} \\
\quad \text{endif} \\
\quad \text{list1} = \text{end1} \\
\text{while Not(IsEmpty(\text{end1}))}
\]

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

- \( \text{cons(var@VariableStar[...],tail)} \rightarrow \)
- \( \text{conc(X*,f(v),Y*)} \preccurlyeq t \rightarrow \text{action} \) is compiled into:

```plaintext
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)
endwhile
endwhile
```
Example of generated code

```java
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);
    }
}
```

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