

Contents/Objectives of the lecture

- Definition and properties of rewriting
- Rule-based programming in ELAN
- Logic and calculus for rewriting
- **Compilation or how to get efficiency?**
- Applications and future developments

Rule-based deduction and computation

Compilation

Hélène Kirchner and Pierre-Etienne Moreau
LORIA – CNRS – INRIA
Nancy, France

Introduction

Overview

- What does compilation mean?
- Compiling syntactic matching
- Compiling associative-commutative matching
- Implementing backtracking
- Compiling rules and strategies
- Deterministic analysis
- Experiments

Overview

- What does compilation mean?
- **Compiling syntactic matching**
- Compiling associative-commutative matching
- Implementing backtracking
- Compiling rules and strategies
- Deterministic analysis
- Experiments

Overview

- What does compilation mean?
- Compiling syntactic matching
- **Compiling associative-commutative matching**
- Implementing backtracking
- Compiling rules and strategies
- Deterministic analysis
- Experiments

Overview

- What does compilation mean?
- Compiling syntactic matching
- Compiling associative-commutative matching
- **Implementing backtracking**
- Compiling rules and strategies
- Deterministic analysis
- Experiments

Overview

- What does compilation mean?
- Compiling syntactic matching
- Compiling associative-commutative matching
- Implementing backtracking
- **Compiling rules and strategies**
- Deterministic analysis
- Experiments

Overview

- What does compilation mean?
- Compiling syntactic matching
- Compiling associative-commutative matching
- Implementing backtracking
- Compiling rules and strategies
- **Deterministic analysis**
- Experiments

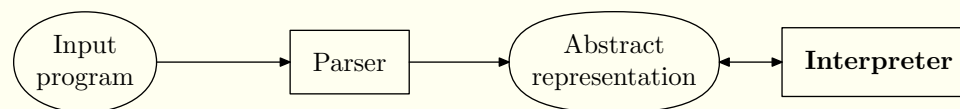
Overview

- What does compilation mean?
- Compiling syntactic matching
- Compiling associative-commutative matching
- Implementing backtracking
- Compiling rules and strategies
- Deterministic analysis
- **Experiments**

What does compilation mean?

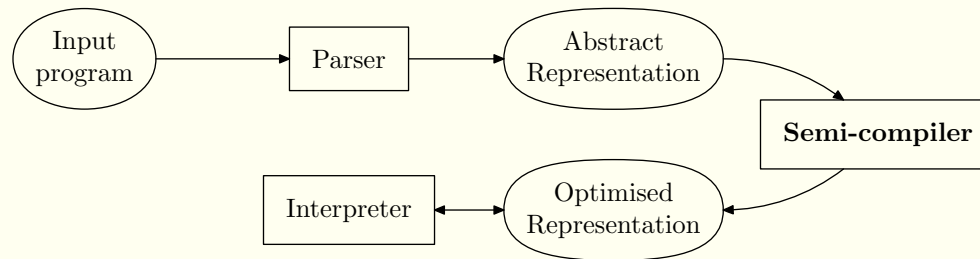
Implementation of a language

- there is at least two possibilities to implement a language on a computer
 - build an **interpreter**
 - build a **compiler**
- An interpreter (for a language L) is a tool that reads a program p_L , some data e , and returns a result or an error:
- $I_L : L \times D^* \mapsto D^* \cup \{error\}$
- $I_L(p_L, e) = r$

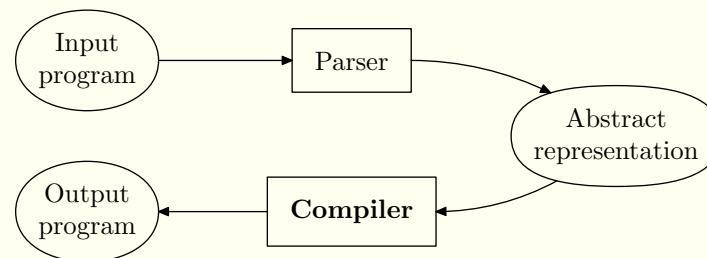


Semi-compiler and compiler

- A **semi-compiler** computes an intermediate optimised representation before doing interpretation

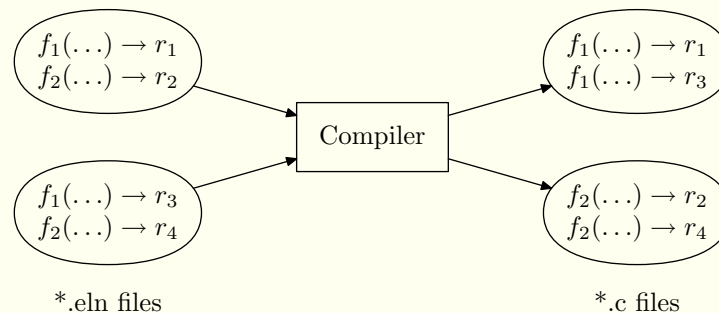


- A **compiler** translates a program into another one. The resulting program is then interpreted or compiled again



A compiler for ELAN

- input programs are **sets of rewrite rules**
- output programs are **C functions** that implement the leftmost-innermost normalisation (wrt. rewrite systems)



Compiling syntactic matching

Syntactic matching

- Problem

- given a subject: $f(g(a), g(c))$
- given a set of rules:
 - * $f(a, g(a)) \rightarrow a$
 - * $f(g(b), g(b)) \rightarrow c$
 - * $f(x, g(c)) \rightarrow b$

- Question

- How to select the rule to apply ?

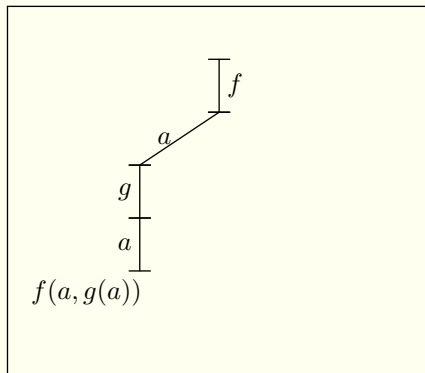
Syntactic matching

- Problem

- given a subject: $f(g(a), g(c))$
- given a set of rules:
 - * $f(a, g(a)) \rightarrow a$
 - * $f(g(b), g(b)) \rightarrow c$
 - * $f(x, g(c)) \rightarrow b$

- Question

- How to select the rule to apply ?



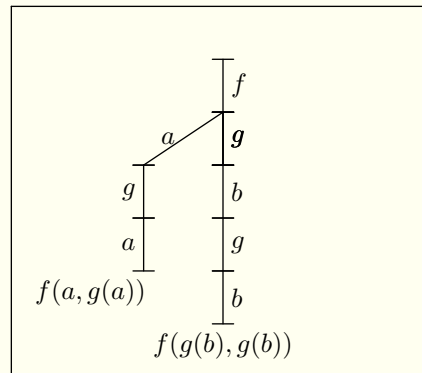
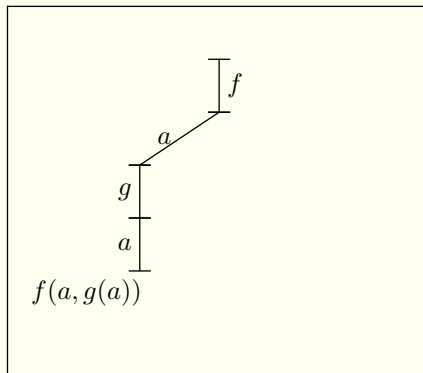
Syntactic matching

- Problem

- given a subject: $f(g(a), g(c))$
- given a set of rules:
 - * $f(a, g(a)) \rightarrow a$
 - * $f(g(b), g(b)) \rightarrow c$
 - * $f(x, g(c)) \rightarrow b$

- Question

- How to select the rule to apply ?



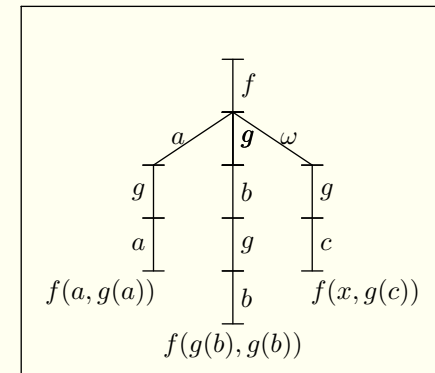
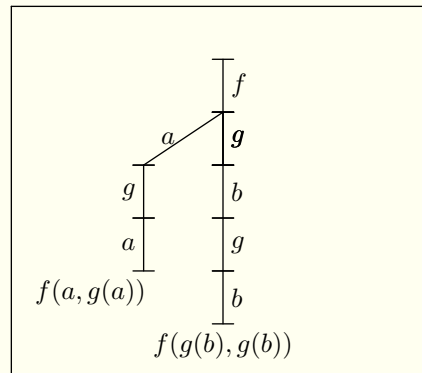
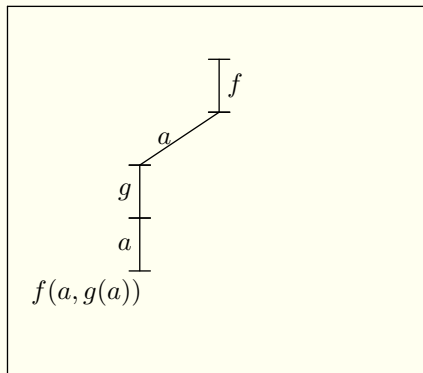
Syntactic matching

- Problem

- given a subject: $f(g(a), g(c))$
- given a set of rules:
 - * $f(a, g(a)) \rightarrow a$
 - * $f(g(b), g(b)) \rightarrow c$
 - * $f(x, g(c)) \rightarrow b$

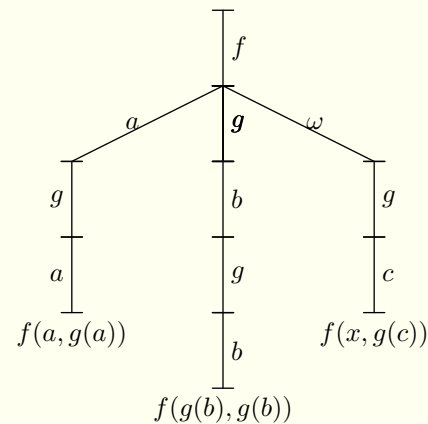
- Question

- How to select the rule to apply ?



Using an automaton

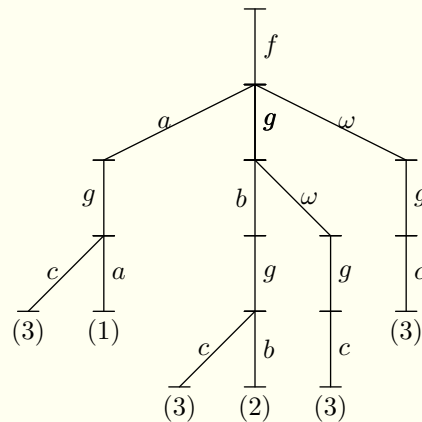
- subject: $f(g(a), g(c))$



- non-determinism: to find a rule to apply
- non-determinism: to try all possibilities

Using a deterministic automaton

- $f(g(a), g(c))$

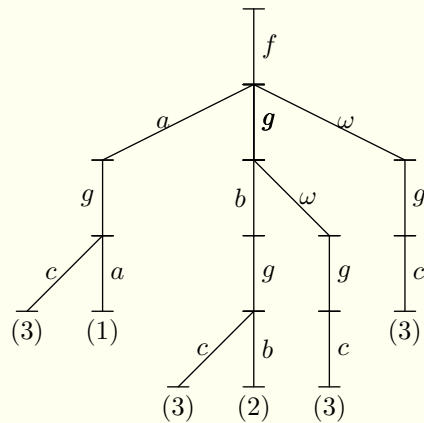


1. $f(a, g(a)) \rightarrow a$
2. $f(g(b), g(b)) \rightarrow c$
3. $f(x, g(c)) \rightarrow b$

Building a deterministic automaton

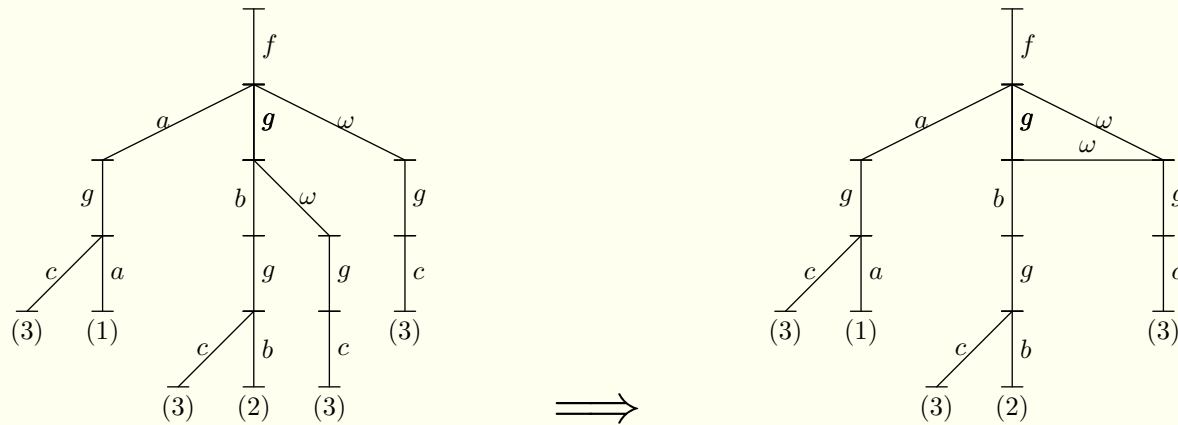
- See Hoffmann and O'Donnell [HO82], A. Gräf [Grä91], Sekar et al. [SRR92], P. Graf [Gra96], Nedjah et al. [NWE97], etc.
- main idea: compute a closure and build the automaton
 - $f(a, g(a))$
 - $f(g(b), g(b))$
 - $f(x, g(c))$
 - $f(a, g(c))$
 - $f(g(x), g(c))$
 - $f(g(b), g(c))$
- problem: contains many states
- optimisations: select a specific traversal-order, share some states, etc.

Example of optimisation



- Use [JumpNode](#) to reduce the size

Example of optimisation



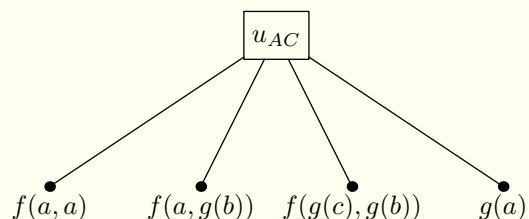
- Use [JumpNode](#) to reduce the size

Compiling associative-commutative matching

Problem to solve

- given a ground term

$$s = u_{AC}(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a))$$



- given a rewrite rule system

$$\begin{aligned} u_{AC}(z, f(a, x), g(a)) &\Rightarrow r_1(z, x) \quad \mathbf{if} \quad z = x \\ u_{AC}(f(a, x), f(y, g(b))) &\Rightarrow r_2(x, y) \end{aligned}$$

- compute a normal form (in an efficient way): a term that cannot be reduced wrt. the rewrite system

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- **build a substitution**
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- **apply a rule**
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Main algorithm

- select a rule
- compute an AC match [HullotThesis-80]
- build a substitution
- check conditions satisfiability
- ask for another match
- apply a rule
- compute a canonical form (an ordered normal form) [HullotThesis-80,Eker-95]

Questions

- Are these steps difficult to implement?
- Where is the problem?

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation
 - designing a many-to-one AC matching algorithm to minimise the cost of the selecting phase

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation
 - designing a **many-to-one AC** matching algorithm to minimise the cost of the selecting phase
 - handle several AC problems in parallel to extract solutions on demand

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation
 - designing a **many-to-one AC** matching algorithm to minimise the cost of the selecting phase
 - handle several AC problems in parallel to extract solutions on demand
 - **maintaining terms in canonical form by construction**

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation
 - designing a **many-to-one AC** matching algorithm to minimise the cost of the selecting phase
 - handle several AC problems in parallel to extract solutions on demand
 - maintaining terms in canonical form by construction
 - **building a compiler rather than an interpreter**

Questions

- Are these steps difficult to implement?
- Where is the problem?
 - combining all these steps together to achieve an efficient implementation
 - designing a **many-to-one AC** matching algorithm to minimise the cost of the selecting phase
 - handle several AC problems in parallel to extract solutions on demand
 - maintaining terms in canonical form by construction
 - building a **compiler** rather than an interpreter

Many-to-one AC matching algorithm

[BachmairChenRamakrishnan1993TAPSOFT]

Given a set of patterns P ,

1. transform rules to fit into a specific class of patterns.
2. compute the AC discrimination net associated to P and the corresponding matching automata.

Given a subject s (ground term),

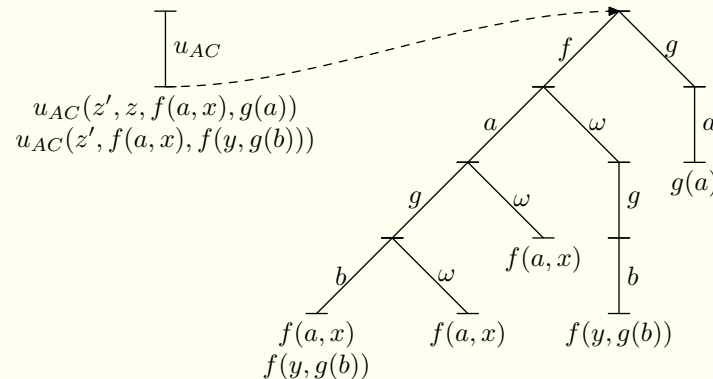
3. build the **hierarchy of bipartite graphs** according to the given subject s in canonical form, and solve it.
4. construct a **Diophantine equational system** to encode the constraints on the remaining unbound variables and solve it.

Example of discrimination net

given a rewrite system:

$$\begin{aligned} u_{AC}(z, f(a, x), g(a)) &\Rightarrow r_1(z, x) \quad \mathbf{if} \quad z = x \\ u_{AC}(f(a, x), f(y, g(b))) &\Rightarrow r_2(x, y) \end{aligned}$$

a **discrimination net** can be built:



Compiling the deterministic discrimination tree

```
int match_subterm_F(struct term *subject, int *mask) {
    switch(getSymb(subject)) {
    case code_g: successor_g=subject->subterm[0];
        switch(getSymb(successor_g)) {
        case code_a:
            mask[nb_bit++]=1;           // g(a)
            break;
        }
        break;
    case code_f: successor_f=subject->subterm[0];
        switch(getSymb(successor_f)) {
        case code_a: successor_a=subject->subterm[1];
            switch(getSymb(successor_a)) {
            case code_g: successor_g=successor_a->subterm[0];
                switch(getSymb(successor_g)) {
                case code_b:
```

```

        mask[nb_bit++]=0;           // f(a,x)
        mask[nb_bit++]=2;          // f(y,g(b))
        break;
    default: goto label7;
}
break;
default:
label7:
    mask[nb_bit++]=0;             // f(a,x)
}
...
}
return nb_bit;
}

```

Example of bipartite graph

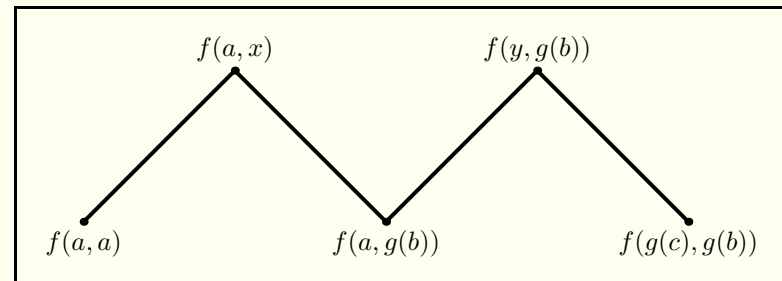
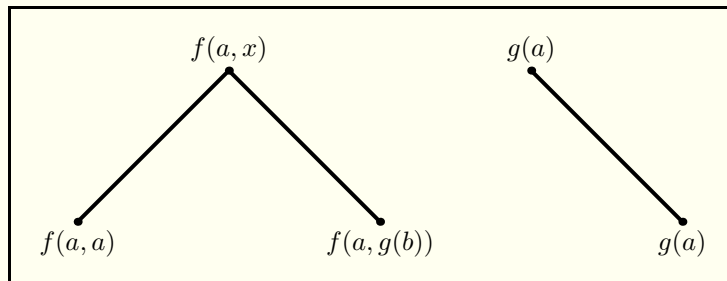
Rules:

$$\begin{aligned} u_{AC}(z, f(a, x), g(a)) &\Rightarrow r_1(z, x) \quad \mathbf{if} \quad z = x \\ u_{AC}(f(a, x), f(y, g(b))) &\Rightarrow r_2(x, y) \end{aligned}$$

Subject:

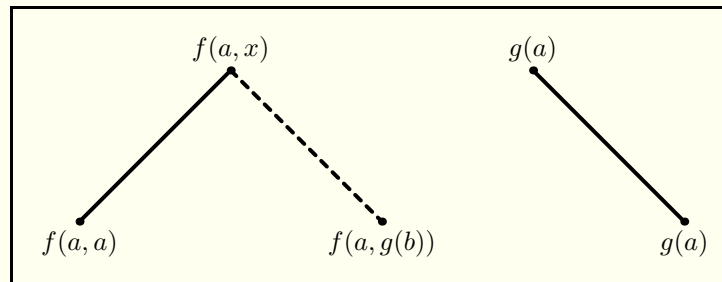
$$s = u_{AC}(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a))$$

two bipartite graphs can be built (one for each rule)



How to apply a rule?

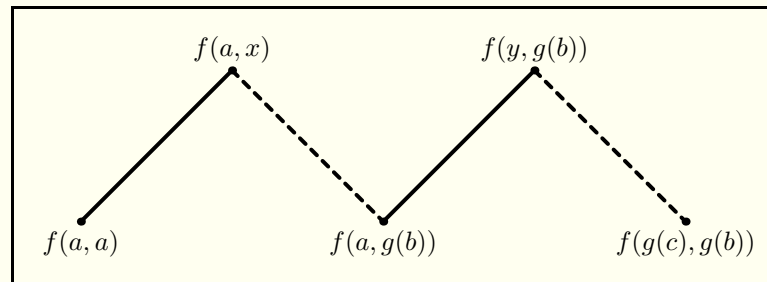
- select a rule (the first one)
- solve the associated bipartite graph: $S = \{f(a, x) \mapsto f(a, a), g(a) \mapsto g(a)\}$



- check the conditions ($x = z$)
- ask for another substitution

How do we continue?

- select another rule (the second one)
- solve the bipartite graph: $S = \{f(a, x) \mapsto f(a, a), f(y, g(b)) \mapsto f(a, g(b))\}$



Our algorithm

- **Patterns are restricted** to at most two levels of AC function symbols (this corresponds to practical cases).
- A new **Compact representation of Bipartite Graphs**, which encodes, in only one data structure, all matching problems relative to the given set of rewrite rules.
- **No Diophantine equational system** is generated when there is at most one or two variables, with (restricted) multiplicity, under an AC function symbol in the patterns.
- A **preliminary syntactic analysis** of rewrite rules can determine that only one solution of an AC matching problem has to be found to apply a rule.

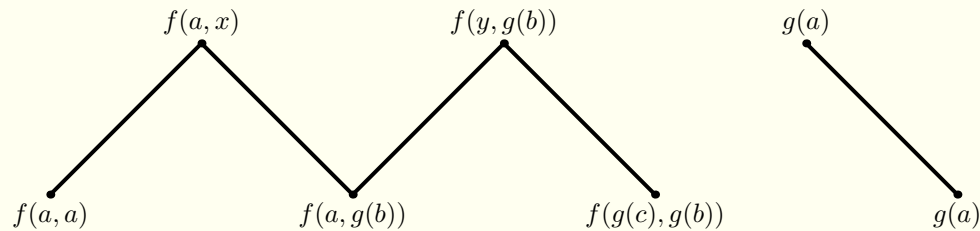
Compact Bipartite Graph (CBG)

Rules:

$$\begin{aligned} u_{AC}(z, f(a, x), g(a)) &\Rightarrow r_1(z, x) \quad \mathbf{if} \quad z = x \\ u_{AC}(f(a, x), f(y, g(b))) &\Rightarrow r_2(x, y) \end{aligned}$$

Subject:

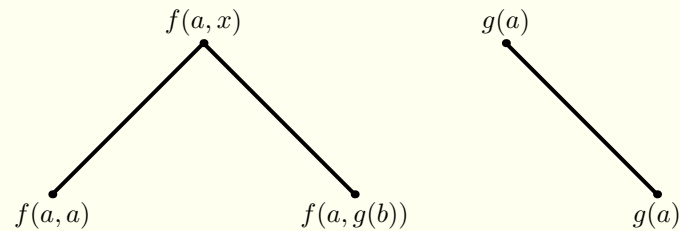
$$s = u_{AC}(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a))$$



Extracted Bipartite Graph

A bipartite graph for the first rule:

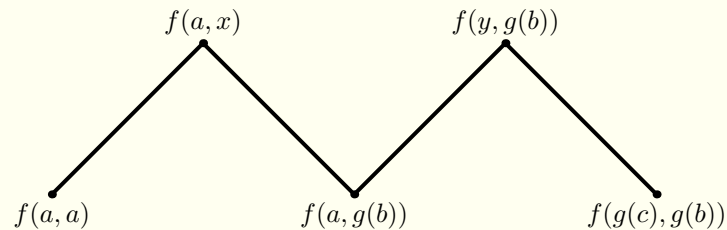
$$u_{AC}(z, f(a, x), g(a)) \Rightarrow r_1(z, x) \text{ if } z = x$$



Two solutions: $(x \mapsto a)$ and $(x \mapsto g(b))$.

Trying another rule

- a new BG is extracted from the CBG
- a bipartite graph for $u_{AC}(f(a, x), f(y, g(b))) \Rightarrow r_2(x, y)$



- no matching phase is done when trying another rule

Compiling the normalisation process

```
struct term* normalise_F(struct term *subject ) {
    struct term *res;
    match_state *ms=NULL;
    /* Begin syntactical matching */
    bitSet32_set(mask32,0);
    bitSet32_set(mask32,1);
    /* Begin AC matching */
    MS_init(&ms, match_subterm_F, pattern_list_F);
    ...
    if(bitSet32_get(mask32,1)) {
        struct term *substitution[3];
        /* lhs: F(z',f(y,g(b)),f(a,x)) */
        substitution_build(subject,ms,substitution,variable_extract_F,1);
        /* rhs: F(z',h(x,y)) */
        if(!isMonoColor(substitution[0])) {
            substitution[0]=normalise_F( substitution[0] );
        }
    }
}
```

```

}
TERM_ALLOC(node_h,code_h);
node_h->subterm[0] = substitution[2];           // x
node_h->subterm[1] = substitution[1];           // y
TERM_ALLOC(node_F,code_F);
term_add_cf_term_color(node_F,substitution[0],color1); // z'
term_add_cf_term_color(node_F,node_h           ,color2); // h(x,y)
res = normalise_F( node_F );
goto end;
} else { ... }
...
match_fail:
  res=subject;
end:
  return res;
}

```

Handling general rules (1)

By program transformation:

- Pattern with an AC top symbol

$$l = u_{AC}(x_1^{\alpha_1}, \dots, x_m^{\alpha_m}, t_1, \dots, t_k, t_{k+1}, \dots, t_n)$$

with $x_1, \dots, x_m \in \mathcal{X}$, $t_1, \dots, t_k \in C_1$ and $t_{k+1}, \dots, t_n \notin C_1$ ($k < n$).

$$l' = u_{AC}(x_1^{\alpha_1}, \dots, x_m^{\alpha_m}, t_1, \dots, t_k, y)$$

The rule

$$l' \Rightarrow r \text{ where } u_{AC}(t_{k+1}, \dots, t_n) := y$$

is equivalent to the previous one.

Example

- the following rule cannot be directly compiled:

$$P + ((p_1 + p_2) * (p_3 + p_4)) \Rightarrow P + e_1 + e_2 + e_3 + e_4$$

where $e_4 := (\text{simplify})p_2 * p_4$
...

- it is automatically transformed into:

$$P + (A_1 * A_2) \Rightarrow P + e_1 + e_2 + e_3 + e_4$$

where $(p_1 + p_2) := ()A_1$
where $(p_3 + p_4) := ()A_2$
where $e_4 := (\text{simplify})p_2 * p_4$
...

Variable instantiation

- For $u_{AC}(x_1, t_1, \dots, t_n)$, once t_1, \dots, t_n are matched, all the unmatched subject subterms are captured by x_1 .
- For $u_{AC}(x_1^{\alpha_1}, x_2, t_1, \dots, t_n)$, once t_1, \dots, t_n are matched, one tries to find in all possible ways α_1 identical remaining subjects to match x_1 and then, all the remaining unmatched subject subterms are captured by x_2 .
- For $u_{AC}(x_1^{\alpha_1}, \dots, x_m^{\alpha_m}, t_1, \dots, t_n)$, once t_1, \dots, t_n are matched, a system of Diophantine equations is solved for computing instances of x_1, \dots, x_m .

Optimisations

- compilation of access functions in order to build the substitution
- allowing destructive update when dealing with un-shared AC terms
- computing normalised substitutions to reduce the number of further rewrite steps
- using tags to avoid unnecessary re-normalisation
- maintaining canonical forms by incrementally flattening terms and merging subterms
- generating an *eager* algorithm when only one AC match is needed (unconditional rules for example)

Experimental Results: Prop

$and(x, \top)$	$\Rightarrow x$	
$and(x, \perp)$	$\Rightarrow \perp$	$xor(x, \perp) \Rightarrow x$
$and(x, x)$	$\Rightarrow x$	$xor(x, x) \Rightarrow \perp$
$and(x, xor(y, z))$	$\Rightarrow xor(and(x, y), and(x, z))$	
$implies(x, y)$	$\Rightarrow not(xor(x, and(x, y)))$	
$not(x)$	$\Rightarrow xor(x, \top)$	
$or(x, y)$	$\Rightarrow xor(and(x, y), xor(x, y))$	
$iff(x, y)$	$\Rightarrow not(xor(x, y))$	

Query

Normalise

implies(and(iff(iff(or(a₁, a₂), or(not(a₃), iff(xor(a₄, a₅), not(not(not(a₆))))))), not(and(and(a₇, a₈), not(xor(xor(or(a₉, and(a₁₀, a₁₁)), a₂), and(and(a₁₁, xor(a₂, iff(a₅, a₅))), xor(xor(a₇, a₇), iff(a₉, a₄))))))))) , implies(iff(iff(or(a₁, a₂), or(not(a₃), iff(xor(a₄, a₅), not(not(not(a₆))))))), not(and(and(a₇, a₈), not(xor(xor(or(a₉, and(a₁₀, a₁₁)), a₂), and(and(a₁₁, xor(a₂, iff(a₅, a₅))), xor(xor(a₇, a₇), iff(a₉, a₄))))))))) , not(and(implies(and(a₁, a₂), not(xor(or(or(xor(implies(and(a₃, a₄), implies(a₅, a₆)), or(a₇, a₈)), xor(iff(a₉, a₁₀), a₁₁)), xor(xor(a₂, a₂), a₇)), iff(or(a₄, a₉), xor(not(a₆), a₆)))))), not(iff(not(a₁₁), not(a₉))))))), not(and(implies(and(a₁, a₂), not(xor(or(or(xor(implies(and(a₃, a₄), implies(a₅, a₆)), or(a₇, a₈)), xor(iff(a₉, a₁₀), a₁₁)), xor(xor(a₂, a₂), a₇)), iff(or(a₄, a₉), xor(not(a₆), a₆)))))), not(iff(not(a₁₁), not(a₉))))))

Bool3

$x + 0$	$\Rightarrow x$	$x * 0$	$\Rightarrow 0$
$x + x + x$	$\Rightarrow 0$	$x * x * x$	$\Rightarrow x$
$(x + y) * z$	$\Rightarrow (x * z) + (y * z)$	$x * 1$	$\Rightarrow x$
$and(x, y)$	$\Rightarrow (x * x * y * y) + (2 * x * x * y) + (2 * x * y * y) + (2 * x * y)$		
$or(x, y)$	$\Rightarrow (2 * x * x * y * y) + (x * x * y) + (x * y * y) + (x * y) + (x + y)$		
$not(x)$	$\Rightarrow (2 * x) + 1$		
2	$\Rightarrow 1 + 1$		

Query

Normalise and compare

$and(and(and(a_1, a_2), and(a_3, a_4)), and(a_5, a_6))$

and

$not(or(or(or(not(a_1), not(a_2)), or(not(a_3), not(a_4))), or(not(a_5), not(a_6))))$

Nat10

In [ContejeanMR-RTA97], a rewrite system modulo AC for natural arithmetic was presented: Nat10.

This system contains
56 rules rooted by the AC symbol $+$,
11 rules rooted by the AC symbol $*$,
and 82 syntactic rules.

A good example showing usefulness of many-to-one matching.

Compute the 16th Fibonacci number.

Experimental Results

	Prop		Bool3		Nat10		Sum100	
	rwr	sec	rwr	sec	rwr	sec	rwr	sec
CiME	-	-	?	> 24h	?	294	-	-
RRL	?	> 24h	?	> 4h	-	-	-	-
Spike	?	> 24h	?	> 24h	-	-	?	> 24h
OBJ	12,837	1,164	?	> 24h	26,936	111	?	> 24h
Brute	23,284	1.78	34,407	2.25	26,648	0.36	177,595	6.25
Maude	12,281	0.47	4,854	0.15	25,314	0.17	177,252	16.77
ELAN	12,689	0.43	5,282	0.18	15,384	0.15	177,152	1.32

Implementing backtracking

Motivations

- design a compiler for ELAN: a rewrite rule based language with strategies

```
[extract] elt.list => elt
```

```
[extract] elt.list => list
```

- a strategy is built on labelled rules: `repeat(dk(extract)) ;`
- `repeat(dk(extract)) (a.b.c)` yields the set `{a, b, c}` ;
- a strategy `S` is compiled into a C function `str_S` ;
- results are returned `on demand`.

Requirements

- we need basic choice point primitives for C programming;
- **setChoicePoint**: create a choice point and save the execution environment;
- **fail**: backtrack to the last created choice point and restore the saved environment;
- these functions can remind the standard C function `setjmp` and `longjmp`;
- **setChoicePoint** and **fail** do not have their limitations.

An example of use in C

Program

```
static int counter=0;
main() {
    if(setChoicePoint()!=0) exit(0);
    f();
    fail();
}
f() {
    int result, locvar=0;
    result=setChoicePoint();
    printf(result,locvar,counter);
    locvar++; counter++;
    printf(locvar,counter);
}
```

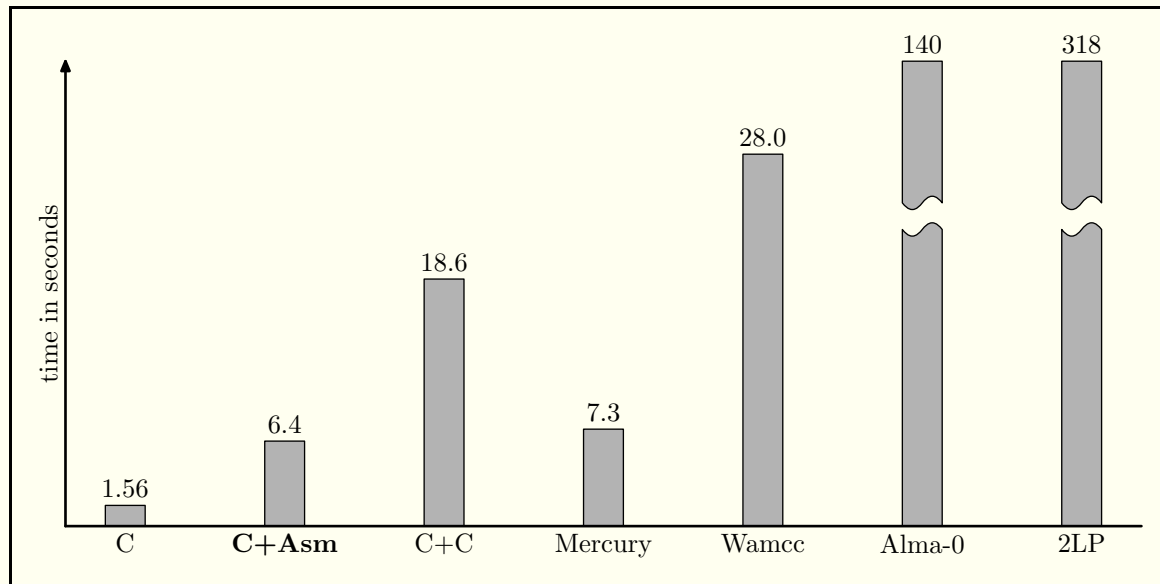
Results

```
result=0, locvar=0, counter=0
locvar=1, counter=1
result=1, locvar=0, counter=1
locvar=1, counter=2
```

A friendly library

- can be directly used by a human (thanks to readability);
- may be used to compile prolog language for example;
- useful to design compilation schemes for new languages that involves nondeterministic computations:
 - complex nondeterministic strategies can be defined in ELAN;
 - 2LP and Alma-0 combine advantages of logic and imperative programming: no needed abstract machine.

Experimental results: 12-queens problem



- C+Asm: assembly “lazy” implementation of `setChoicePoint` and `fail`
- C+C: using the C version implemented with `setjmp` and `longjmp`

Concluding remarks

- “plug-in” design: function calls, local variables and modular compilation are possible;
- readable code: useful to design complex compilation schemes;
- the assembly code can be “easily” ported (Sparc, Alpha, i86);
- a good compromise between simplicity and efficiency;
- available (under GPL) at <http://www.loria.fr/ELAN/Toolkit>

Compiling rules and strategies

The evaluation mechanism

- unlabelled rules are applied with a leftmost-innermost strategy
- a rewrite rule has the following form:

$$\begin{aligned} [\ell] \quad f(x) \rightarrow & h(y_3, y_4) \\ & \text{where } y_1 := ()g(x) \\ & \text{where } y_2 := (Strat_1)g(x) \\ & \text{if } y_1 == y_2 \\ & \text{where } f(y_3, y_4) := (Strat_2)g(x) \end{aligned}$$

- conditional rules: the condition must be a **boolean** term
- **backtracking** may occur in local assignment evaluation

Basic strategy operators

- a labelled rule is a strategy
- ℓ applied on t returns all terms reduced by the labelled rule ℓ
- basic constructors for rules and strategies:
 - `one`(S): returns only 1 result
 - `all`(S): returns all possible results
- selectors for strategies:
 - `select_one`(S_1, \dots, S_n) = S_i such that S_i does not fail
 - `select_all`(S_1, \dots, S_n) returns all strategies

Builtin strategies in ELAN

$dc_one(S_1, \dots, S_n)$	$=$	$select_one(one(S_1), \dots, one(S_n))$
$dc(S_1, \dots, S_n)$	$=$	$select_one(all(S_1), \dots, all(S_n))$
$dk(S_1, \dots, S_n)$	$=$	$select_all(all(S_1), \dots, all(S_n))$

- remark that: $dc(S) = dk(S) = S$
- ‘;’ to compose two strategies
- **id**: does nothing (but does not fail)
- **fail**
- **repeat**: applies a strategy as long as possible up to a failure
- **iterate**: same as repeat but returns all intermediate results

the N-queens problem

```
rules for list
  [queens_n] queens(i,N) => queens . queensList
    if i>0
      where queensList:=(queens_strat) queens(i-1,N)
      where queen:=(n_to_1) N
      if noattack(queen,queensList)
    end
strategies for list
  [] queens_strat      => dk(queens_n)      end
  [] det_queens_strat => dc_one(queens_n)  end
strategies for int
  [] n_to_1  => iterate(dc(range_rule))  end
```

Compilation of unlabelled rules

- unlabelled rules beginning with the **same top symbol** are compiled into one C function
- a **deterministic matching automaton** is generated
- if a rule can be applied, the right-hand side is built (in a bottom-up way):
 - **variables** are built (with sharing)
 - **constants** are built (with sharing)
 - **constructors** are allocated (without sharing)
 - **defined symbols** are built by calling the corresponding normalisation system (i.e. the set of rules with the same top-symbol)

Compilation of local evaluations

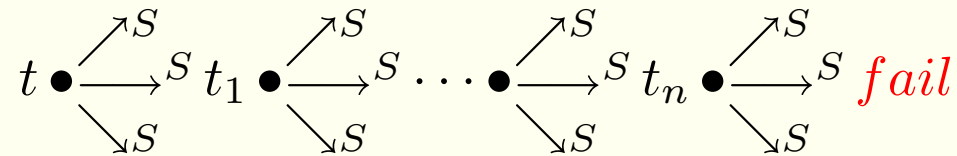
Before compiling the right-hand side, local evaluations are compiled:

- **if** *cond*
 - 1: $c \leftarrow$ evaluation of *cond*
 - 2: **if** $c \neq \text{true}$ **then fail**
- **where** $p := (S)t$
 - 1: $t' \leftarrow$ evaluation of S applied to t
 - 2: one-to-one matching from p to t'
 - 3: **if** matching fails **then fail**

compilation of strategies (the idea)

- each ELAN strategy is compiled into a C function
 - a matching automaton is generated
 - `setChoicePoint` saves the environment,
 - `fail` goes back to the last `ChoicePoint`
 - `cutOpen` and `cutClose` are used to delete set `ChoicePoint`
- compilation of `dk`: for each strategy and applicable rule, a `ChoicePoint` is set before its application.
- compilation of `dc_one` begins with a `cutOpen` and ends with a `cutClose`
- compilation of composition, `id` and `fail` is simple
- when matching phase fails, a `fail` is generated

Compilation of repeat



- one **ChoicePoint** per step is needed
- when a failure occurs: one **ChoicePoint** is deleted and the process continues
- if S returns at most one result: **ChoicePoint** are not necessary
- **setChoicePoint** and **fail** are useful to get a readable generated code and design some new compilation schemes

Conclusion

- compilation of rules and strategies is not so complex
- we only need to know how to:
 - generate a (associative-commutative) matching automaton
 - set a choice-point
 - remove a choice point and backtrack
- in general, it may be inefficient:
 - **too many choice-point** are used
 - **too much memory** is allocated
- **solution**: perform static analysis

Deterministic analysis

Determinism mode

- at compile time, a determinism mode can be inferred:

det:	one result (never fails)
semi:	zero or one result
multi:	more than one result (never fails)
nondet:	more than zero result
fail:	no result

determinism analysis algorithm

- $d\text{-m}(\text{one}(S)) = \begin{cases} \mathbf{det} & \text{if } d\text{-m}(S) \text{ is } \mathbf{det} \text{ or } \mathbf{multi} \\ \mathbf{semi} & \text{if } d\text{-m}(S) \text{ is } \mathbf{semi} \text{ or } \mathbf{nondet} \end{cases}$
- $d\text{-m}(\text{repeat}(S)) = \begin{cases} \mathbf{det} & \text{if } d\text{-m}(S) \text{ is } \mathbf{det} \text{ or } \mathbf{semi} \\ \mathbf{multi} & \text{if } d\text{-m}(S) \text{ is } \mathbf{multi} \text{ or } \mathbf{nondet} \end{cases}$
- the `repeat` operator cannot fail
- $d\text{-m}(\text{iterate}(S)) = \mathbf{multi}$
- a default mode is inferred when the recursivity problem occurs

determinism analysis algorithm (cont.)

- $d\text{-m}(\text{select_one}(S_1, \dots, S_n)) = \text{And}(d\text{-m}(S_1), \dots, d\text{-m}(S_n))$

<i>And</i>	det	semi	multi	nondet	fail
det	det	semi	multi	nondet	fail
semi	semi	semi	nondet	nondet	fail
multi	multi	nondet	multi	nondet	fail
nondet	nondet	nondet	nondet	nondet	fail
fail	fail	fail	fail	fail	fail

- $d\text{-m}(\text{select_all}(S_1, \dots, S_n)) = \text{Or}(d\text{-m}(S_1), \dots, d\text{-m}(S_n))$

Impact of determinism analysis

- improved compilation scheme for **det** or **semi** strategies
- only one **ChoicePoint** is needed to compile `repeat(semi)`
 - • $t \xrightarrow{S} t_1 \xrightarrow{S} \dots \xrightarrow{S} t_n \xrightarrow{S} fail$
 - set single **ChoicePoint**
 - apply the strategy S as many times as possible and store each result in *lastTerm*
 - when a failure occurs: the **ChoicePoint** is deleted and the saved term *lastTerm* is returned

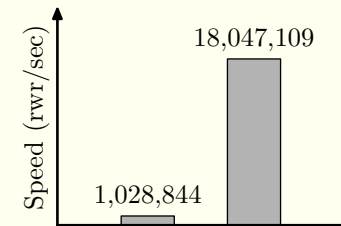
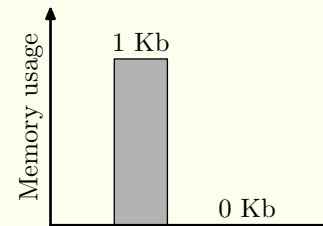
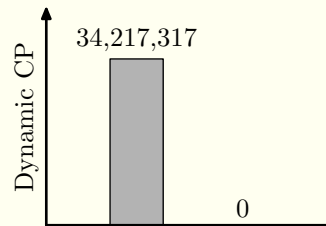
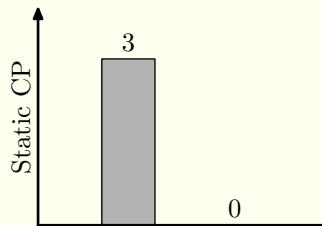
Impact of determinism analysis (cont.)

- the search space, memory usage, number of **ChoicePoint**, and time spent in backtracking and memory management can be considerably reduced
- detect some non-terminating strategies (**repeat(det)** for example)
- improve the efficiency of AC matching: only one solution has to be extracted for **semi** or **det** rules

Experiments

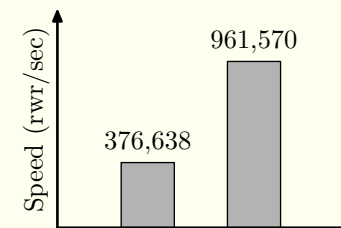
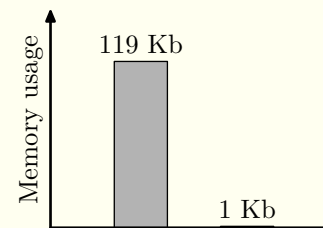
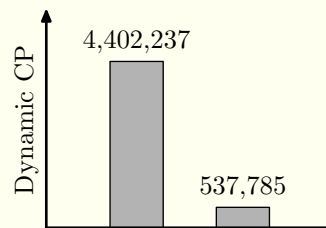
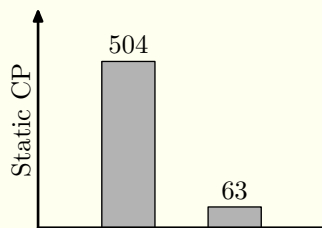
Impact of deterministic analysis: Fibonacci

- query: fib(33)
- a functional program
- no choice-point should be used



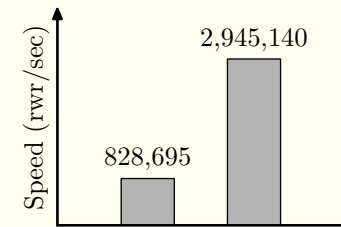
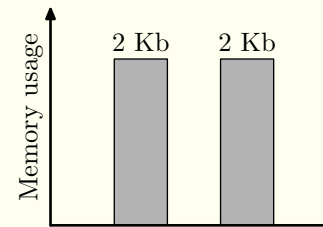
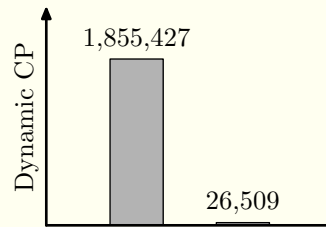
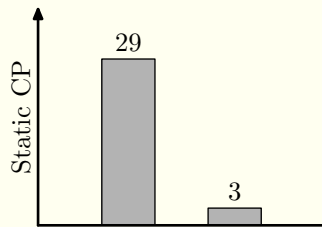
Impact of deterministic analysis: Knuth-Bendix Completion

- query: p5 (group + 5 neutral elements + 5 inverse)
- a large non-deterministic program



Impact of deterministic analysis: N-Queens

- query: nqueens(14)



Experimental results

- 1 to 2 millions rewrite rules per second
- from 30,000 to 100,000 pure AC rewrite step per second
- up to 15 millions for very simple examples

time (sec)	OCaml	GProlog	Elan	OBJ	Brute
Fibonacci(35)	1.80	-	2.97	-	-
Nqueens(12)	19.0	57.0	31.2	-	-
Prop	-	-	0.43	1,164	1.78
Bool3	-	-	0.59	> 24h	6.18
Sum100	-	-	0.41	> 24h	6.25

Why it is efficient?

- defined symbols are compiled into function calls
- many-to-one pattern matching (deterministic automata)
- simple (but frequent) AC patterns are compiled efficiently [ALP/PLILP'98]
 - no recursive call
 - carefully designed Compact Bipartite Graph structures
 - no general diophantine equation
- deterministic analysis [WADT'98]
 - less choice points and memory usage
 - more updates in place
 - better AC matching algorithms

Concluding remarks

Old Compiler [1996]

- written in C++ (builtin tool)
- efficient but limited
- good prototype

-
- ELAN is available at:
www.loria.fr/ELAN

New Compiler [2000]

- written in Java (REF tool)
- modular compilation
- new many-to-one matching
- AC matching
- builtin sorts
- deterministic analysis
- full ELAN

References

- [Grä91] A. Gräf. Left-to-right tree pattern matching. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 323–334. Springer-Verlag, April 1991.
- [Gra96] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.
- [HO82] C. M. Hoffmann and M. J. O’Donnell. Pattern-matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.
- [NWE97] N. Nedjah, C.D. Walter, and E. Eldrige. Optimal left-to-right pattern-matching automata. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Proceedings 6th International Conference on Algebraic and Logic Programming, Southampton (UK)*, volume 1298 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, September 1997.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In W. Kuich, editor, *Proceedings of ICALP 92*, volume 623 of *Lecture Notes in Computer Science*, pages 247–260. Springer-Verlag, 1992.

To know more:

- ESSL'2001 notes

- “Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories”.

H.Kirchner, P.E.Moreau *J.Functional Programming*, 11(2):207-251, March 2001.