

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 computation and deduction

Applications and extensions

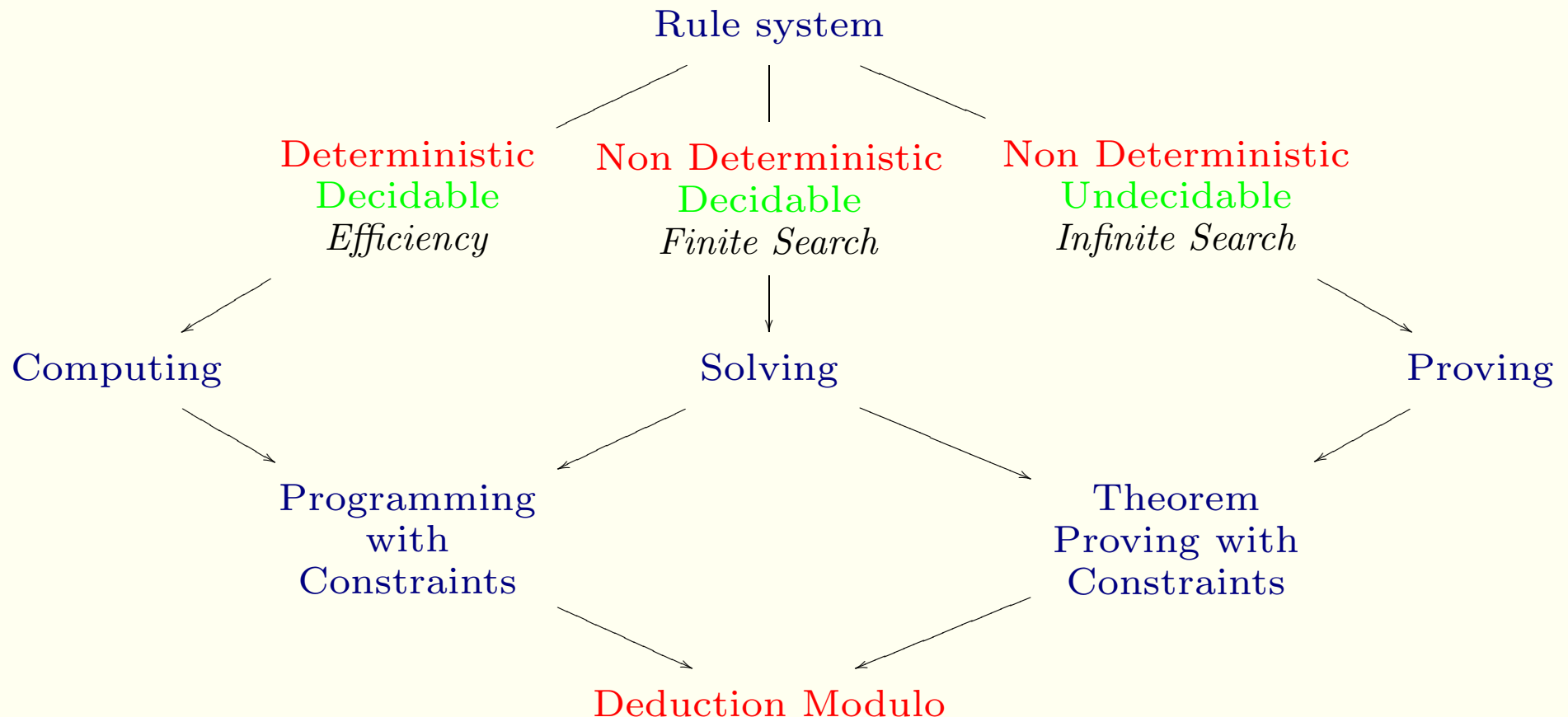
Hélène Kirchner and Pierre-Etienne Moreau
LORIA – CNRS – INRIA
Nancy, France

Introduction

Overview

- Rule-based programming: computation and deduction in application
- How to use rule-based techniques in various environments?
- How to build a new environment ? Adding objects in ELAN
- Other typical problems of the field

Rules for computing, solving, proving



Rewriting for computing

$$\frac{\log(x \times y) \rightarrow \log(x) + \log(y) \quad \mathbf{if} \quad x \geq 0 \wedge y \geq 0}{}$$

$$\frac{\begin{array}{l} |x| \rightarrow x \quad \mathbf{if} \quad x \geq 0 \\ |x| \rightarrow -x \quad \mathbf{if} \quad x < 0 \end{array}}{}$$

$$\begin{array}{l} fib(0) \rightarrow 1 \\ fib(1) \rightarrow 1 \\ fib(n) \rightarrow fib(n-1) + fib(n-2) \quad \mathbf{if} \quad n > 1 \end{array}$$

Rewriting for solving

Equation solving

Does it exist x, y, z such that:

$$x + (z * y) = y + (x * z)$$

There is an infinity of solutions

but a most general one

$$x = y = z$$

====> syntactic unification

The rules for syntactic unification

Delete	$P \& s =? s$	
	$\rightarrow P$	
Decompose	$P \& f(s_1, \dots, s_n) =? f(t_1, \dots, t_n)$	
	$\rightarrow P \& s_1 =? t_1 \& \dots \& s_n =? t_n$	
Conflict	$P \& f(s_1, \dots, s_n) =? g(t_1, \dots, t_p)$	
	$\rightarrow \mathbf{F}$	if $f \neq g$
Coalesce	$P \& x =? y$	
	$\rightarrow \{x \mapsto y\}P \& x =? y$	if $x, y \in \mathcal{V}ar P \wedge x \neq y$

Check*	$P \ \& \ x_1 =^? s_1[x_2] \ \& \ \dots$ $\dots \ \& \ x_n =^? s_n[x_1]$	
	$\rightarrow \mathbf{F}$	if $s_i \notin \mathcal{X}$ for some $i \in [1..n]$
Merge	$P \ \& \ x =^? s \ \& \ x =^? t$	
	$\rightarrow P \ \& \ x =^? s \ \& \ s =^? t$	if $0 < s \leq t $
Check	$P \ \& \ x =^? s$	
	$\rightarrow \mathbf{F}$	if $x \in \mathcal{V}ar \ s$ and $s \notin \mathcal{X}$
Eliminate	$P \ \& \ x =^? s$	
	$\rightarrow \{x \mapsto s\}P \ \& \ x =^? s$	if $x \notin \mathcal{V}ar \ s, s \notin \mathcal{X}, x \in \mathcal{V}ar \ P$

Example

$$x + (z * y) = y + (x * z)$$

$$\rightarrow_{\text{decompose}} x = y \ \& \ z * y = x * z$$

$$\rightarrow_{\text{decompose}} x = y \ \& \ z = x \ \& \ y = z$$

$$\rightarrow_{\text{coalesce}} y = z \ \& \ x = z \ \& \ z = x$$

$$\rightarrow_{\text{coalesce}} z = x \ \& \ y = x \ \& \ x = x$$

$$\rightarrow_{\text{delete}} z = x \ \& \ y = x$$

Rewriting for Proving

Completion procedure:

Allows to relate

- Equational deduction with a set of axioms E
- Rewrite deduction with a set of rules R

From a set of axioms, deduce an equivalent set of rewrite rules

deduction rules for completion.

computations: order terms, compute critical pairs, normalise.

Simplification orderings

A **simplification ordering** is an irreflexive transitive relation $>$ on terms, closed under context and substitution, that contains the subterm ordering.

R is **simply terminating** if there exists a simplification ordering $>$ such that for any rule $l \rightarrow r$ in R , $l > r$.

When the set \mathcal{F} of operator symbols is finite, a rewrite system R is terminating if R is simply terminating [DershowitzTCS-1982].

Simplification orderings can be built from a well-founded ordering on the function symbols \mathcal{F} called a **precedence**.

Example: lexicographic path ordering.

Lexicographic path ordering LPO

Let $>_F$ be a precedence on F .

$$s = f(s_1, \dots, s_n) >_{lpo} t = g(t_1, \dots, t_m)$$

if one at least of the following conditions holds:

1. $f = g$ and $(s_1, \dots, s_n) >_{lpo}^{lex} (t_1, \dots, t_m)$ and $\forall j \in \{1, \dots, m\}, s >_{lpo} t_j$
2. $f >_F g$ and $\forall j \in \{1, \dots, m\}, s >_{lpo} t_j$
3. $\exists i \in \{1, \dots, n\}$ such that either $s_i >_{lpo} t$ or $s_i = t$.

LPO in ELAN

```
[] lpo(s,t) => true
  if not(isvar(s)) and not(isvar(t))
  choose
  try    if head(s) ==sig head(t)
        where b:= () lpo3(s,t)
        if b
  try    if head(s) >sig head(t)
        where b:= () lpo2(s,t)
        if b
  end
end
end
```

Critical Pairs

Superposition

$$\begin{array}{l} l_1 \longrightarrow r_1 \qquad l_2[u] \longrightarrow r_2 \\ l_2[r_1]\sigma = r_2\sigma \end{array}$$

u is a non-variable sub-term of l_2
 σ is the $mgu(u, l_1)$ (i.e. $u\sigma = l_1\sigma$)

All critical pair should satisfy:

$$l_2[r_1]\sigma \xrightarrow{*}_R \circ R \xleftarrow{*} r_2\sigma$$

The saturation rules

Orient	$P \cup \{p = q\}, R$	\mapsto	$P, R \cup \{p \rightarrow q\}$ if $p > q$
Deduce	P, R	\mapsto	$P \cup \{p = q\}, R$ if $(p, q) \in CP(R)$
Simplify	$P \cup \{p = q\}, R$	\mapsto	$P \cup \{p' = q\}, R$ if $p \rightarrow_R p'$
Delete	$P \cup \{p = p\}, R$	\mapsto	P, R
Compose	$P, R \cup \{l \rightarrow r\}$	\mapsto	$P, R \cup \{l \rightarrow r'\}$ if $r \rightarrow_R r'$
Collapse	$P, R \cup \{l \rightarrow r\}$	\mapsto	$P \cup \{l' = r\}, R$ if $l \xrightarrow[R]{g \rightarrow d} l'$ and $l \rightarrow r \gg g \rightarrow d$

The main result

The sets of persisting rules and pairs are defined as:

$$P_\infty = \bigcup_{i \geq 0} \bigcap_{j > i} P_j \quad \text{and} \quad R_\infty = \bigcup_{i \geq 0} \bigcap_{j > i} R_j.$$

If the derivation $(P_0, R_0) \mapsto (P_1, R_1) \mapsto \dots$ satisfies

- all critical pairs have been computed ($CP(R_\infty)$ is a subset of $\bigcup_{i \geq 0} P_i$),
- R_∞ is reduced and
- P_∞ is empty,

then R_∞ is Church-Rosser and terminating.

Moreover $\xrightarrow{*} P_0 \cup R_0$ and $\xrightarrow{*} R_\infty$ coincides.

An additive group G is defined by the set of equalities

$$\begin{array}{lcl} x + e & = & x \\ x + (y + z) & = & (x + y) + z \\ x + i(x) & = & e \end{array} \quad \left| \quad \begin{array}{lcl} x + e & \rightarrow & x \\ e + x & \rightarrow & x \\ x + (y + z) & \rightarrow & (x + y) + z \\ x + i(x) & \rightarrow & e \\ i(x) + x & \rightarrow & e \\ i(e) & \rightarrow & e \\ (y + i(x)) + x & \rightarrow & y \\ (y + x) + i(x) & \rightarrow & y \\ i(i(x)) & \rightarrow & x \\ i(x + y) & \rightarrow & i(y) + i(x) \end{array}$$

How to use rule-based techniques for combining deduction and computation in various environments?

How to use rule-based techniques for combining deduction and computation in various environments?

- Encode your problem in ELAN

How to use rule-based techniques for combining deduction and computation in various environments?

- Encode your problem in ELAN
- Design by yourself a rewrite engine in your system

How to use rule-based techniques for combining deduction and computation in various environments?

- Encode your problem in ELAN
- Design by yourself a rewrite engine in your system
- Connect an existing rewrite tool and trust or check its results

How to use rule-based techniques for combining deduction and computation in various environments?

- Encode your problem in ELAN
- Design by yourself a rewrite engine in your system
- Connect an existing rewrite tool and trust or check its results
- Use TOM a pattern matching preprocessor

Rewriting techniques in automated theorem provers

- **Boolean algebras and rings** Applications to proof search in first order logic (Hsiang, 1985).
- **Proof of commutativity in specific rings**

$$(\forall x, x^n = x) \Rightarrow \forall x, y, (x * y = y * x)$$

$n = 3$ (Stickel, 1984), n even (Kapur,Zhang, 1991).

- **The Robbins conjecture**

In a Boolean algebra

$$\overline{\overline{x + y} + \overline{x + y}} = y$$

implies

$$\overline{\overline{x + \overline{y}} + \overline{x + \overline{y}}} = y$$

(McCune, 1996)

ELAN for equational reasoning in CoQ

Efficient tools to implement decision procedures using rewriting for CoQ.

- ELAN is used as a rewrite engine to find a computation.
- Reflection is used in CoQ to translate back the delegated rewrite computation.
- A trace is returned by ELAN and replayed by CoQ, to ensure correctness and build a complete proof term.

Examples: Groups, rings,...Stalmarck's algorithm.

The Common Framework Initiative for Algebraic Specifications

CoFI is a collaborative effort involving many different groups working on algebraic specifications.

CASL is a general-purpose specification language from which a family of related specification languages can be obtained by restriction, or by extension, all with a consistent, user-friendly syntax and clear semantics.

Provide tools for **CASL** or reuse existing ones.

Executing Casl specifications with ELAN

A class of structured conditional equational specifications of $CASL$.

A translation that makes use of the related abstract syntaxes:

- Casfix: abstract syntax for $CASL$ generated by $CASL$ parsers.
- Efix: abstract syntax for $ELAN$ generated by a new $ELAN$ parser.

Casfix and Efix are two similar instances of $ATerms$

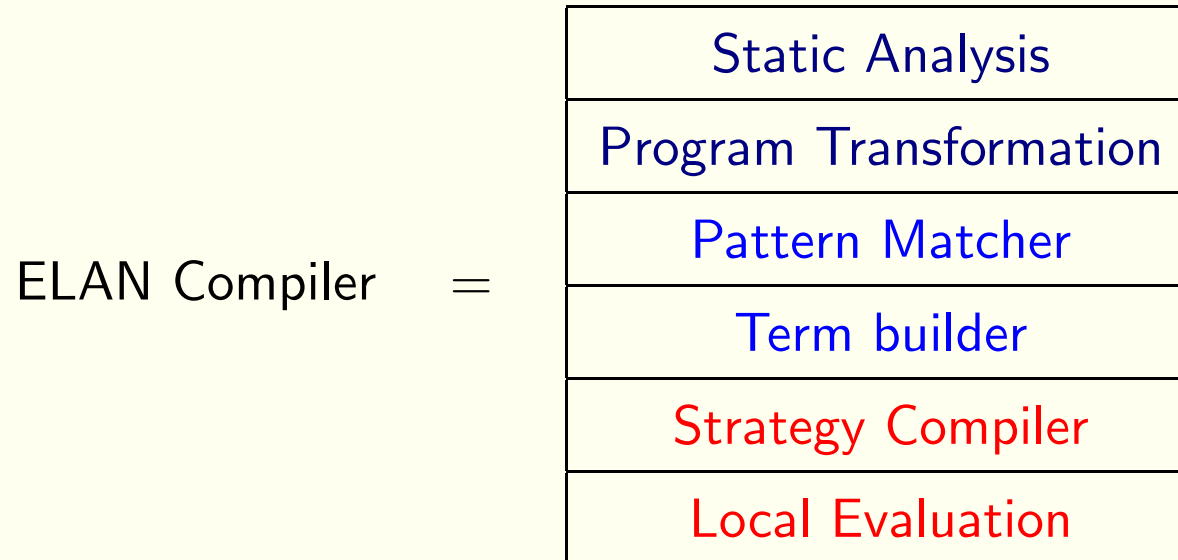
A 2-steps translation $Casfix \rightarrow Efix \rightarrow REF$
implemented using the $ATerms$ library

TOM: a pattern matching preprocessor

ELAN → E.L.A.N

From a rule-based programming language
to a pattern matching preprocessor

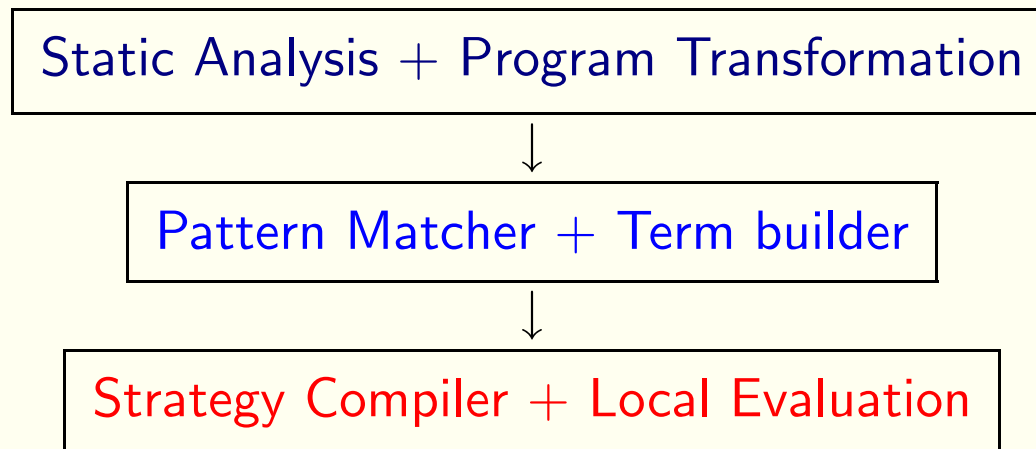
The current environment



Drawback: parts are difficult to reuse

The future environment

Idea: decompose the system into reusable parts



TOM \approx Pattern Matcher + Term builder

Motivations

- **TOM** should not be connected to ELAN

but

- **TOM** should be usable to implement ELAN
- **TOM** should be usable to perform term rewriting

Main Ideas

- Be able to program with rules in:
 - Imperative languages: C, Java, C++
 - Computer Algebra: MuPad
 - Constraint programming: Claire
 - etc.
- Without using a fixed (built-in) term data structure
 - Terms should be user-defined
- Underlying technology: compilation of discrimination trees, Many-to-One matching, One-to-One matching, etc.

Potential Applications

- implementation of rule based engines (ELAN, ASF+SDF, ...)
- ruled-based computation for Computer Algebras Systems
- program transformation (software reengineering)
- XML document processing

Exercise

- How implementing the following rewrite system in Java?

$$\begin{aligned} \textit{plus}(x, 0) &\rightarrow x \\ \textit{plus}(x, \textit{suc}(y)) &\rightarrow \textit{suc}(\textit{plus}(x, y)) \\ \textit{fib}(0) &\rightarrow \textit{suc}(0) \\ \textit{fib}(\textit{suc}(0)) &\rightarrow \textit{suc}(0) \\ \textit{fib}(\textit{suc}(\textit{suc}(n))) &\rightarrow \textit{plus}(\textit{fib}(n), \textit{fib}(\textit{suc}(n))) \end{aligned}$$

Target Language Implementation

- Class definition, Symbols definition (using ATerm from CWI), ...

```
import jaterm.api.*;
import jaterm.shared.*;
```

```
public class Jfib {
    AFun symbol_zero;
    AFun symbol_suc;
    AFun symbol_plus;
    AFun symbol_fib;
    ...
}
```

API Definition

(**term** \iff **ATerm**)

```
%typeterm term {  
  implement      { ATerm }  
  get_fun_sym(t) { (((ATermAppl)t).getAFun()) }  
  get_subterm(t, n) { (((ATermAppl)t).getArgument(n)) }  
  equals(t1, t2)  { (t1.equals(t2)) }  
}
```

```
%op term zero      { fsym { symbol_zero } }  
%op term suc(term) { fsym { symbol_suc  } }  
%op term plus(term,term) { fsym { symbol_plus } }  
%op term fib(term)  { fsym { symbol_fib  } }
```

Rule Definition

```
%rule plus(x, zero)    -> { return x;          }
%rule plus(x, suc(y))  -> { return suc(plus(x,y)); }
```

- equivalent to

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

Another Definition

```
%rule fib(zero)          -> { return suc(zero);          }
%rule fib(suc(zero))     -> { return suc(zero);          }
%rule fib(suc(suc(x)))  -> { return
                           plus(fib(x),fib(suc(x))); }
```

```
ATerm fib(ATerm t) {
  %match(term t) {
    zero          -> { return suc(zero));          }
    suc(zero)     -> { return suc(zero));          }
    suc(suc(x))  -> { return plus(fib(x),fib(suc(x))); }
  }
}
```

Some functions

```
void symbolicFib(int n) {
    ATerm t = zero;
    for(int i=0 ; i<n ; i++) {
        t = suc(t);
    }
    t = fib(t);
    ...
}

public final static void main(String[] args) {
    ... symbolicFib(10); ...
}
```

Syntax Summary

TOM program = Target Language + TOM constructs

- API Definition for user-defined term data structure
 - sort mapping (`get_fun_sym`, `get_subterm`, ...)
 - operator mapping (`fsym`, ...)
- Rule Definition (`%rule`)
- Match Construct (`%match`)

TOM Features

- **Built-in types:** integers, floats, strings
 - Placeholders are allowed: "_"
 - Terms can be named: $f(x:g(y)) \rightarrow \{ \dots \ x \ \dots \ }$
 - useful for memory management
- ```
%rule plus(x, arg:plus(y,z)) -> {
 free(arg); return(plus(plus(x,y),z)); }
```

# Extensions

- Present:
  - List-matching
  - Matching automata: smaller and more efficient
  - Term Rewrite Rules (a right-hand side defined by a Term)
- Future:
  - Other equational theories (AC,...)
  - Strategy constructors (Non-deterministic, Traversal, etc.)
  - Automatic verification on rewrite systems

# Conclusion

- TOM is **easy** to use
- you can use **your** term data structure
- it will integrate some **efficient** matching algorithms (with compact generated automata)
- it will integrate some **expressive** matching algorithms: associative and commutative operators, neutral element, ...
- it **is** available at <http://elan.loria.fr/Toolkit>

# Adding objects to ELAN

# Introduction

Design of an environment for production rules systems

- with rules of the form

**if** *conditions* **then** *actions*

- applying on a data base of objects

Oz [Smolka et al.,95], ILOG Rules [97], CLAIRE [Caseau et al.,94]

# What means: adding objects to ELAN?

An extension of the language:

- Which syntax ?

# What means: adding objects to ELAN?

An extension of the language:

- Which syntax ?
  - Main concepts: classes, objects, methods, inheritance
  - Rules applying on objects to express changes of objects

# What means: adding objects to ELAN?

An extension of the language:

- Which syntax ?
  - Main concepts: classes, objects, methods, inheritance
  - Rules applying on objects to express changes of objects
- Which operational semantics ?

# What means: adding objects to ELAN?

An extension of the language:

- Which syntax ?
  - Main concepts: classes, objects, methods, inheritance
  - Rules applying on objects to express changes of objects
- Which operational semantics ?
  - based on  $\rho$ -calculus

## Example

```
class MLift

imports ToolsMLift

attributes CF:int = 0
 State:LiftState = Wait
 LStop:list[int] = nil
 Zone:int = 0
 F:int = 0
 I:int = 0

method WhichSense(N:int) for MLift
 S : Sense;
 <S:=ChooseSense(self.GetCF,N) ; self.SetState(Move(S))>
... End
```

# Algebraic Representation of objects - I

- Object structure defined by constants and operators
- Methods are defined by functions and rewrite rules

## Algebraic Representation of objects - II

Declarations of operators to encode and manipulate objects :

|                 |   |                                    |           |           |      |
|-----------------|---|------------------------------------|-----------|-----------|------|
| $[-]$           | : | $Methods$                          | $\mapsto$ | $Object$  |      |
| $-, -$          | : | $Methods \times Methods$           | $\mapsto$ | $Methods$ | (AC) |
| $-$             | : | $Method$                           | $\mapsto$ | $Methods$ |      |
| $-(\_)$         | : | $MName \times MBody$               | $\mapsto$ | $Method$  |      |
| $-$             | : | $MName$                            | $\mapsto$ | $Method$  |      |
| $add(\_, -)$    | : | $Object \times Method$             | $\mapsto$ | $Object$  |      |
| $kill(\_, -)$   | : | $Object \times MName$              | $\mapsto$ | $Object$  |      |
| $access(\_, -)$ | : | $Object \times MName$              | $\mapsto$ | $MBody$   |      |
| $Get(\_, -)$    | : | $Object \times MName$              | $\mapsto$ | $MBody$   |      |
| $Set(\_, -, -)$ | : | $Object \times MName \times MBody$ | $\mapsto$ | $Object$  |      |
| $new(\_)$       | : | $Object$                           | $\mapsto$ | $Object$  |      |

## $\mathcal{R}$ : a rewrite system for objects

- $\mathcal{R}$  contains for instance a rule

- to add a new method :

$$add([LM], me) \rightarrow_{\mathcal{R}} [LM, me]$$

- to suppress a method

- to access to values, to modify values,

- to create a new object, etc...

- The system  $\mathcal{R}$  is terminating and confluent

# The rewrite system $\mathcal{R}$

|                                     |                                                                                    |
|-------------------------------------|------------------------------------------------------------------------------------|
| <b>Add a component</b>              | $add([LM], me) \rightarrow_{\mathcal{R}} [LM, me]$                                 |
| <b>Suppress a component-1</b>       | $kill([M(B), LM], M) \rightarrow_{\mathcal{R}} [LM]$                               |
| <b>Suppress a component-2</b>       | $kill([M, LM], M) \rightarrow_{\mathcal{R}} [LM]$                                  |
| <b>Access to an attribute value</b> | $access([M(B), LM], M) \rightarrow_{\mathcal{R}} B$                                |
| <b>Access to a value by Get</b>     | $Get(o, M) \rightarrow_{\mathcal{R}} access(o, M)$                                 |
| <b>Modify a value by Set</b>        | $Set(o, M, B) \rightarrow_{\mathcal{R}} add(kill(o, M), M(B))$                     |
| <b>Create a new object</b>          | $new(o) \rightarrow_{\mathcal{R}} [a_1(vi_1), \dots, a_n(vi_n), m_1, \dots, m_m]$  |
| <b>Call a method <math>m</math></b> | $[LM, m].m(p_1, \dots, p_m) \rightarrow_{\mathcal{R}} m([LM, m], p_1, \dots, p_m)$ |

# From Objects to ELAN programs

- Sorts, constants and operators declarations:
  - Each attribute: constant and sort of the attribute
  - Each method: a constant and an operator
- A set of unlabelled rules implementing the system  $\mathcal{R}$
- A set of rules associated to user-defined methods.

## Rules for methods

How to build rewrite rules associated to methods?

*build* – rule( **method** *name* (*args*) **for** *sort vars* *body*) =  
    **rules for** *sort*  
    *S* : *class\_name* ;  
    var-decl(*vars, args, body*)  
    [] *name*(*S, args*) → *build-rhs*(*vars, body*) **end**  
    **end**

*build* – *rhs* builds the right-hand side of the rule.

## An example

```
method WhichSense(N:int) for MLift
 S : Sense;
<S:=ChooseSense(self.GetCF,N) ; self.SetState(Move(S))>
```



```
rules for MLift
 o : MLift;
 N : int;
 S : Sense;
global
[] WhichSense(o,N) => o.SetState(Move(S))
 where S := () ChooseSense(o.GetCF,N)
end
```

## How is it implemented?

Use ELAN to execute object programs:

- Object modules are translated into ELAN modules
- An ELAN program performs the translation

## Rules on objects

$$[lab] O_1 \dots O_k \Rightarrow O'_1 \dots O'_m \text{ [if } t \mid \text{where } l]^*$$

- Objets  $O_i$  are of the form :

$$O_i : ClassName_i :: [Att_1(Value_1) , \dots , Att_n(Value_n)]$$
$$O_i : ClassName_i$$

- Different objects : modified objects, contextuels objects, suppressed objects, new objects

## Applying a rule

$$[lab] \ O_1 \dots O_k \Rightarrow O'_1 \dots O'_m \ [\mathbf{if} \ t \mid \mathbf{where} \ l]^*$$

- Verification of matching conditions in the working memory
- Evaluation of conditions to true
- Replacement of  $O_i$  ( $i = 1, \dots, k$ ) by  $O'_j$  ( $j = 1, \dots, m$ )

## Example

```
Vars 01 : MLift;
 02 : LCall;

[Up] 01:MLift::[State(Move(Up)) , F(0) , I(0)]
 02:LCall
 =>
 01(CF<-01.CF+1).UpdateZone(F<-1)
 02
 if not(in(01.CF,01.Stop))
 if not(in(01.CF,02.LCall))
```

# Operational semantics for ELAN with objects

- Starting points:
  - $\rho$ -calculus defined by H. Cirstea and C. Kirchner.
  - Object  $\rho$ -calculus defined by H. Cirstea, C. Kirchner and L. Liquori [RTA'01]
- Then:
  - mapping terms encoding objects to  $\rho$ -terms with a map  $\tau$
  - proving equivalence of computations.



## $\tau$ : correspondance term/ $\rho$ -term

$\tau$  maps a term  $t_0$  of the form :

$$t_0 = [m(b) , Getm , Setm , LM]$$

to a  $\rho$ -term  $t'_0$  :

$$\begin{aligned} \tau(t_0) = t'_0 = & (m \rightarrow S \rightarrow b, \\ & Getm \rightarrow S \rightarrow S \cdot m, \\ & Setm \rightarrow S \rightarrow N \rightarrow S \cdot m := S' \rightarrow N, \\ & L) \end{aligned}$$

Proposition : For any term  $t$ ,  $\tau(FN(t)) \equiv \tau(t)$ .

## To go further

- Definition of classes: add global/local attributes/methods
- Use strategies in object modules
- Extend the language by adding and dealing with constraints
- Compile the object programs into an internal format directly executable by the interpreter/compiler.

# Conclusion

## Typical problems of the field

Modelizing with rewrite relation

Confluence of  $\mathcal{R}$  (and extensions)

Termination of  $\mathcal{R}$  (and extensions)

Compute normal forms (with strategies)

Find normal forms optimally, when possible

Modularity properties of rewrite systems

## For example for termination

$$f(a, b, x) \rightarrow f(x, x, x)$$

is terminating, as well as

$$g(x, y) \rightarrow x$$

$$g(x, y) \rightarrow y,$$

Is the union terminating?

## Too bad ...

$$\begin{aligned}f(a, b, x) &\rightarrow f(x, x, x) \\g(x, y) &\rightarrow x \\g(x, y) &\rightarrow y,\end{aligned}$$

We have the cycle:

$$\overbrace{f(g(a, b), g(a, b), g(a, b)) \rightarrow f(a, g(a, b), g(a, b)) \rightarrow f(a, b, g(a, b))}$$

[Toyama 1986]

## What about confluence?

[Toyama 1987] proved that

The disjoint union of two confluent rewrite systems is confluent

What happens if the two rewrite systems share symbols?

## To know more

- The rewriting page:

<http://rewriting.loria.fr/>

- The IFIP WG 1.6 working group on rewriting and applications:

- The RTA conferences

- Books:

Rewriting and all that

[Baader Nipkow 1998] (Cambridge University Press)

Rewriting, Solving, Proving

[Kirchner Kirchner 1994-2000] ([www.loria.fr/~ckirchne/rsp.ps.gz](http://www.loria.fr/~ckirchne/rsp.ps.gz))

- Surveyssss

- Slides of ESSLLI lecture will be available at

[www.loria.fr/~hkirchne/](http://www.loria.fr/~hkirchne/) and [www.loria.fr/~moreau/](http://www.loria.fr/~moreau/)

**The End**