Introduction to ELAN and the rewrite calculus

Claude Kirchner
LORIA and INRIA
Nancy

Claude.Kirchner@loria.fr

Companion web site: `www.loria.fr/ELAN`

Companion introduction to the ρ-calculus: `www.loria.fr/~ckirchne/=rho`

Casting: Horatiu Cirstea, Hubert Dubois, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen

Rules are ubiquitous in Mathematics, Logic and Computer Science

Production rules

Program transformation rules

Grammar rules

Transition rules

Logic programming rules

Constraint manipulation rules

Inference rules

Type checking rules

Derivative rules

and much more ...

Some examples

- the Unix mail system: look at the /etc/sendmail.cf file!
- tree automaton
- the so called *natural semantics*
- Mathematica
- Reve, Spass, Otter, Saturate
- ...

Some success

- re-engineering of Cobol programs [ASF+SDF, CWI Amsterdam]
- proof of the Robbin's conjecture [New York Times, December 1996]
  In a Boolean algebra,
  $$\overline{\overline{\overline{x} + y} + \overline{x + y}} = y$$
  implies
  $$\overline{\overline{x} + \overline{y}} + \overline{x + \overline{y}} = y$$
  (McCune, 1996)
- . . .

. . . and ELAN

Two main kinds of rules

Computation rules

$2 + 3$

fib(33)

sort(data-base)

Compute as fast as possible the unique result

Deduction rules

Sequent calculus

Resolution method

Solve($2x + 4y - 3z - u = 0$ in $\mathbf{N}$)

The deduction has to be guided

Deduction rule application needs to be controlled

Lazy evaluation

Innermost/Outermost reduction

Search plans

Action plans

Tactics

User interaction

▶ This requires to search for a particular derivation corresponding to the desired control.

rewrite rewrite rewrite rewrite rewrite rewrite rewrite rewrite

Logic Programming, Theorem Proving, Constraint Solving, Expert Systems are instances of the same deduction schema:

$$\boxed{\text{Apply rewrite rules on formulas with some strategy, until getting specific forms}}$$

- Rewrite blindly: implements computations
- Rewrite wisely: implements deduction

Poincaré principle [BarendregtBarendsen97]

Deduction modulo [DowekHardinKirchner98]

How this presentation will proceed:

— Give a first introduction of ELAN with an emphasis on control of rewriting

— Present the rewriting calculus as a semantical foundation for controled rule based systems

— Come back on ELAN and its main features

Short presentation of ELAN

ELAN= computation rules + (deduction rules + strategies)

ELAN's General features

- ELAN has been designed to prototype, experiment and study deduction systems, in particular for constraint solving and theorem proving.
- ELAN is a declarative rewriting language.
- ELAN has two main originalities w.r.t. other algebraic languages:
  - non-deterministic computations
  - a user defined strategy language to control rewriting

Deductions and computations

- Rules for computations:
  - unique normal form required
  - leftmost innermost deterministic strategy fixed
- Rules for deductions:
  - no confluence nor termination required
  - application strategy required.
    - non-determinism handled in practice by backtracking
    - strategies are used to express choices
    - strategy may fail

Example 1: Very simple ...

```
module fib_builtin
import global builtinInt;
  end

operators global
        fib(@)  : (builtinInt) builtinInt ;
  end

rules for builtinInt
        n : builtinInt ;
global
    [] fib(0) => 1 end
    [] fib(1) => 1 end
    [] fib(n) => fib(n - 1) + fib(n - 2) if greater_builtinInt(n,1) end
  end
end
```

fib(33) = 5702887   11405773 rewrite steps in 0.695 s   16.411.184 rewrite/s

Digital 500/500, 128Mo

Example 2: Polynome derivative in ELAN

```
module poly1       import global int;    sort poly;
operators global
  X             : poly;
  @             : ( int ) poly;
  @ + @         : ( poly poly ) poly  assocRight;
  deriv(@)      : ( poly ) poly;
end
rules for poly
  p1, p2 : poly;  n : int;
global
  []  deriv(X)      => 1                   end
  []  deriv(n)      => 0                   end
  []  deriv(p1+p2)  => deriv(p1)+deriv(p2)  end
end
```

```
module poly3[Vars]
import global int Vars identifier list[identifier];
sort variable poly;
operators global
  FOR EACH Id:identifier SUCH THAT
    Id:=(listExtract) elem(Vars) : { Id : variable; }
  @           : ( variable )  poly;
  @           : ( int )       poly;
  @ + @       : ( poly poly ) poly  assocRight pri 1 (AC);
  (@ + @)     : ( poly poly ) poly  alias @ + @:;
  @ * @       : ( poly poly ) poly  assocRight pri 2 (AC);
  (@ * @)     : ( poly poly ) poly  alias @ * @:;
  deriv(@,@) : ( poly variable )   poly;
end

... ... ...
```

A simple strategy

```
[]  P+0                              => P                end
[]  P*0                              => 0                end
[]  deriv(x,x)                       => 1                end
[]  deriv(n,x)                       => 0                end
[]  deriv(p1+p2,x)        => deriv(p1,x) + deriv(p2,x)   end

[factorize]  P + n1*p1 + n2*p1       => P + n3*p1
                                        where n3:=()n1+n2 end
[factorize]  P + (p1*p2) + (p1*p3)   => P + p1*(p2+p3)   end
[expand]     P + (p1+p2)*p3          => P + p2*p3 + p1*p3 end
end
strategies for poly
  [] simplify  =>
    normalize( one(expand) ) ; normalize( one(factorize) )
  end
```

Elementary strategies

Built from a few built-in constructors:

- Sequential composition

  $S_1; S_2$

- Iteration

  $\mathsf{repeat*}(S)$ or $\mathsf{iterate*}(S)$

- Choice points

  $\mathbf{dk}(S_1, \ldots, S_n)$: all results of all S_i

  $\mathbf{dc}(S_1, \ldots, S_n)$: all results of one S_i

  $\mathbf{first}(S_1, \ldots, S_n)$: all results of the first S_i

- Cut points

  $\mathsf{dc\ one}(S_1, \ldots, S_n)$: one result of one S_i

  $\mathsf{first\ one}(S_1, \ldots, S_n)$: one result of the first S_i

- Failure **fail** and Identity **id**

Another example: propositional sequent calculus

$$\frac{H, P \vdash Q}{H \vdash \neg P, Q}\text{neg-r}$$

```
rules for Seq
        P, Q, R : Pred;   H : Pred;   S1, S2 : Seq;
global
  [neg-r]     H |- ^P : Q    => S1
                                        where S1 := (dedstrat) H : P |- Q
  end

...

  [axio]   P : H |- P : R   =>  $
                                    if neq_Pred(P, EmptyP)
```

The true code

Built (for later use) the proof term:

```
[negr]    H |- ^P : Q    =>  [#negd,H |- ^P : Q] <> S1
                                where S1 := (dedstrat) H : P |- Q
end
```

Strategies

```
strategies  for Seq
implicit

        [] SetRules  =>  first one(
                                axio
                                ,negd ,disjd
                                ,impd ,negg ,conjg
                                ,disjg ,conjd ,impg)
        end
end

strategies  for Seq
implicit
        [] dedstrat  =>  first one( Start );
                         repeat*( SetRules )
        end
end
```

The resulting proof term

```
[dedstrat](A |=> B |- ^(B) |=> ^(A))
```

evaluates to:

```
#infer[#impd]<(A#to B)#vdash(#neg(B)#to#neg(A))>
<#infer[#negd]<(A#to B),#neg(B)#vdash#neg(A)>
<#infer[#negg]<A,(A#to B),#neg(B)#vdash EmptyP>
<#infer[#impg]<A,(A#to B)#vdash B>
<#infer[#axiom]<A,B#vdash B><#mbox<>>&
#infer[#axiom]<A#vdash A,B><#mbox<>>>>>>
end
```

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{A, B \vdash B}\,Axiom \qquad \cfrac{}{A \vdash A, B}\,Axiom}{A, (A \to B) \vdash B}\,\to_G}{A, (A \to B), \neg(B) \vdash}\,\neg_G}{(A \to B), \neg(B) \vdash \neg(A)}\,\neg_D}{(A \to B) \vdash (\neg(B) \to \neg(A))}\,\to_D$$

To summarize: Rules in ELAN

Rules are possibly labeled conditional rewrite rules with local variable affectations

$$[\mathsf{lab}] : l(x) \Rightarrow r(y) \quad \textbf{if } v \quad \textbf{where } y := [S]u(x)$$

— lab is the rule label,

— l and r the respective left and right-hand sides,

— v the condition and

— $y := [S]u$ a local affectation, assigning to the local variable y one of the results of the strategy S applied to the term u.

Computation and Deduction in ELAN

Computations are described with unlabeled rules

Deductions are described by labeled rules

Strategies for deductions are themselves described using labeled or unlabeled rewrite rules.

The Computational System Tower

...

...

Meta Meta Strategies

Meta Strategies

Strategies

Terms

Rewriting with strategies IS now a realistic programming paradigm

On typical applications, ELAN applies per second:

- 10 millions of non-labeled rules
- 1 million of labeled rules
- 100 000 of AC rules

On real size applications:

- Jobs shop 10x10: 2 billions rewrites applied in 2 hours

Some Applications developed in ELAN

- Theorem provers (KBs, Spike, B)
- Automata computations
- CP on finite domains
- Unifications
- Combination of deduction systems
- ...

N-queens

$$
\begin{array}{lll}
queens(n, size) & \rightarrow & x.ql \text{ if } n > 0 \\
& & \text{where } ql := (queens\_strat)queens(n-1, size) \\
& & \text{where } x := (iterate * (dc(range\_rule)))size \\
& & \text{if } noattack(1, x, ql) \\
& \ldots &
\end{array}
$$

	Interpreter	Compiler	Compiler
Query	queens(8)	queens(8)	queens(11)
Applied rules	128,949	128,949	19,286,638
Speed (inf/sec)	6,570	2,387,000	2,743,000

- **Note:** local affectations and non-deterministic strategies
- **Note:** efficient management of non-deterministic strategies
- **Speedup:** 360

Completion à la Knuth Bendix

- 225 nonamed rules
- 105 named rules
- 43 strategies (8 dont know, 58 dont care, 6 iterate, 14 repeat)

	Interpreter	Compiler	Compiler
Query	group	group	p8
Applied rules	181,752	181,752	21,607,279
Speed (inf/sec)	1,452	826,145	919,458

▶ **Note:** conditional rules, local affectations and non-deterministic strategies,. . .

▶ **Speedup:** 135

Nat10

Arithmetic rules introduced by Marché et all

- 56 rules rooted by the AC symbol +
- 11 rules rooted by the AC symbol *
- 82 syntactic rules

	Interpreter	Compiler	Compiler
Query	Fib(16)	Fib(16)	Fib(25)
Applied rules	15,384	15,384	1,171,043
Speed (inf/sec)	44	49,625	53,717

How does it work?

- leftmost-innermost normalization
- many-to-one matching
- re-using left-hand-side parts
- shared terms allowed
- particular choice point management

$\rightarrow$ What about the semantics?

The rewriting calculus as a semantics of ELAN

Towards a new calculus

That gives a first class status to:

- rewrite rules

and

- strategies

The main ideas (1)

apply a rule at the top level of a term

$$[l \rightarrow r](t)$$

The main ideas (2)

The application operator $[\quad](\quad)$ returns results that should be handled explicitely.

For example, if $+$ is commutative, what is the result of the application of the rule

$x + y \rightarrow x$

on

$a + b$

should it be a?
or b?
or $a \diamond b$?

The main ideas (3)

— rule application

— results object

should be explicit objects of the calculus

The calculus ingredients

Five components:

1. The syntax of terms and substitutions,
2. The description of the substitution application on terms,
3. The matching algorithm used to bind variables to their actual arguments,
4. The evaluation rules describing the way the calculus operates locally.
5. The strategy describing how the evaluation rules operate globally.

1 ρ-Calculus Syntax

- elements in $\mathcal{X}$ (variables) and in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ are ρ-terms,

If $t, u, t_1, \ldots, t_m$ are ρ-terms and $f \in \mathcal{F}_m$ then the following are ρ-terms:

- $f(t_1, \ldots, t_m)$
- $\{t_1, \ldots, t_m\}$ (if $m = 0$ we have the ρ-term $\emptyset$),
- $[t](u)$ (application of the ρ-term t to the ρ-term u),
- $t \to u$ (rewrite rule).

$$t ::= x \mid \{t, \ldots, t\} \mid f(t, \ldots, t) \mid [t](t) \mid t \to t \qquad (x \in \mathcal{X},\ f \in \mathcal{F})$$

Examples of ρ-terms

- $f(x, y)$ a first order term
- $f(x, x) \to x$ a "standard" rewrite rule
- $[a \to b](a)$ the application of the rule $a \to b$ to the term a
- $[f(x, y) \to g(x, y)](f(a, b))$ a classical rule application
- $[x \to x](a)$ the result is $\{a\}$
- $[x \to y](a)$ the result is $\{y\}$
- $[x \to (x \to x)]([x \to y](a))$ similar to the λ-term $((\lambda x.\lambda x.x)\ ((\lambda x.y)\ a))$
- $[x \to [x](x)](x \to [x](x))$ the well-known $(\Delta\ \Delta)$ λ-term
- $[(a \to b) \to c](a)$ a more complicated ρ-term
- $[\{a \to b, a \to c\}](a)$ apply several rules

Handling results

Handling results via the set datastructure:

- has consequences on the calculus evaluation rules and their properties
- could be done in a different ways using e.g. list, multisets, ...

2 Substitution application on ρ-terms

Takes care of variable capture.

Grafting	Substitution
$x \mapsto u$	x/u
$\{x \mapsto a + y\}(y \rightarrow y + x) = y \rightarrow y + (a + y)$	$\{x/a + y\}(y \rightarrow y + x) = y' \rightarrow y' + (a + y)$

4 ρ-Calculus: Evaluation Rules

The main rule in general:

$$\textbf{Fire} \quad [l \to r](t) \quad \mapsto\!\!\!\!\twoheadrightarrow \quad r\langle\!\langle \mathcal{S}olution(l \ll^{?}_{T} t)\rangle\!\rangle$$

For example

$$\textbf{Fire} \quad [f(x) \to x](f(a)) \quad \mapsto\!\!\!\!\twoheadrightarrow \quad x\langle\!\langle \mathcal{S}olution(f(x) \ll^{?}_{T} f(a))\rangle\!\rangle$$

In case of syntactic matching:

$$\begin{array}{llll} \textbf{Fire} & [l \to r](\sigma(l)) & \mapsto\!\!\!\!\twoheadrightarrow & \{\sigma(r)\} \\ \textbf{Fire} & [l \to r](t) & \mapsto\!\!\!\!\twoheadrightarrow & \emptyset \\ & & & \text{if } l \text{ does not match } t \end{array}$$

Congruence rules on operators

Congruence	$[f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n))$	$\Vdash\!\!\twoheadrightarrow$	$\{f([u_1](v_1),\ldots,[u_n](v_n))\}$
Congruence_fail	$[f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m))$	$\Vdash\!\!\twoheadrightarrow$	$\emptyset$

Equivalent to the reduction of

$[f(x_1,\ldots,x_n) \to f([u_1](x_1),\ldots,[u_n](x_n))](t)$

Congruence rules on sets

Distrib	$[\{u_1, \ldots, u_n\}](v)$	$\twoheadrightarrow$	$\{[u_1](v), \ldots, [u_n](v)\}$
Batch	$[v](\{u_1, \ldots, u_n\})$	$\twoheadrightarrow$	$\{[v](u_1), \ldots, [v](u_n)\}$
SwitchL	$\{u_1, \ldots, u_n\} \to v$	$\twoheadrightarrow$	$\{u_1 \to v, \ldots, u_n \to v\}$
SwitchR	$u \to \{v_1, \ldots, v_n\}$	$\twoheadrightarrow$	$\{u \to v_n, \ldots, u \to v_n\}$
OpOnSet	$f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n)$	$\twoheadrightarrow$	$\{f(v_1, \ldots, u_1, \ldots, v_n),$ $\ldots,$ $f(v_1, \ldots, u_m, \ldots, v_m)\}$

Handling sets of sets

Flat $\{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \quad \mapsto\!\!\!\twoheadrightarrow \quad \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}$

Applying substitutions

Meta-rules in this version

$$
\begin{array}{lll}
\textbf{Propagate} & r\langle\!\langle \{s_1, \ldots, s_n\} \rangle\!\rangle & \leadsto \quad \{s_1(r), \ldots, s_n(r)\} \\
\textbf{Clash} & r\langle\!\langle \emptyset \rangle\!\rangle & \leadsto \quad \emptyset
\end{array}
$$

Could be made explicit: $\rho\sigma$-calculus

Remarks

- ρ-calculus describes rewrite rules application at the top level of a term

$$[f(x) \to g(x)](f(b)) \underset{Fire}{\mapsto\!\!\!\twoheadrightarrow} \{g(b)\}$$

but

$$[x + 0 \to x](f(3 + 0)) \underset{Fire}{\mapsto\!\!\!\twoheadrightarrow} \emptyset$$

and

$$[f(x + 0 \to x)](f(3 + 0)) \underset{Congruence}{\mapsto\!\!\!\twoheadrightarrow} f([x + 0 \to x](3 + 0)) \underset{Fire}{\mapsto\!\!\!\twoheadrightarrow} f(\{3\})$$

- The evaluation rules expressing the behavior of ρ-calculus are applied *everywhere* in the term (e.g. like β-reduction).

3 Matching

In general

For a theory T a T*-match-equation* is a formula of the form $t \ll^?_T t'$, where t and t' are ρ-terms.

A substitution σ is a T-solution of $t \ll^?_T t'$ when

$$\sigma(t) =_T t'$$

T-matching is in general undecidable.

Decidable cases:

- higher-order pattern matching [Miller-89]
- higher-order matching (up to the fourth order [PadovaniThese-96])
- many first-order equational theories and their combinations

Solutions set

We define the function $\mathcal{S}olution$ on a T-matching system $\mathcal{S}$ as returning the set of:
— all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial
— $\{Id\}$ (the identity substitution) when $\mathcal{S}$ is trivial
— $\emptyset$ when the matching algorithm fails.

Specificities of the ρ_T-calculus

- calculus of explicit rule application
- get the “matching power”
- explicit handling of result sets,
- one of the main parameter of the calculus is the matching algorithm,

- allows the full control of the rewrite rule application
- allows a uniform combination of higher-order and first-order features,
- allows expressing strategies (e.g. search strategies)

- the ρ-calculus is quite different from the rewrite relation generated by a trs.

The $\rho_\emptyset$-calculus

An instance of ρ_T-calculus where:

— $T = \emptyset$
— the left members of matching equations are composed only of first-order terms (i.e. not containing arrows or applications).

From now on we only consider $\rho_\emptyset$-calculus.

No difficulty to extend to the case of first-order equational matching (e.g. AC-matching)

Technical specificities

- A rule application always fires ... but the result may be the empty set (when the rule does not match an under-evaluated term),
- Thus the wild $\rho_\emptyset$-calculus is trivialy non-confluent:

$$\begin{tikzcd} & {[x \to b](\emptyset)} \arrow[dl, "Fire"'] \arrow[dr, "Batch"] & \\ \{b\} & & \emptyset \end{tikzcd}$$

Undesired matching failures

$$\begin{tikzcd}
 & & {[a \to b]([a \to a](a))} \arrow[dll, "Fire"'] \arrow[dr, "Fire"] & \\
{[a \to b](\{a\})} \arrow[d, "Batch"'] \arrow[dr, "Fire"] & & & \emptyset \\
\{[a \to b](a)\} \arrow[d, "Fire"'] & \emptyset & & \\
\{\{b\}\} \arrow[d, "Flat"'] & & & \\
\{b\} & & &
\end{tikzcd}$$

Non-right-linear rewrite rules

$$
\begin{tikzcd}
{[x \rightarrow f(x,x)](\{a,b\})} \arrow[d, "Fire"'] \arrow[dr, "Batch"] & \\
\{f(\{a,b\},\{a,b\})\} \arrow[d, "OpOnSet"'] & \{[x \rightarrow f(x,x)](a), [x \rightarrow f(x,x)](b)\} \arrow[d, "Fire"] \\
\{\{f(a,\{a,b\}), f(b,\{a,b\})\}\} \arrow[d, "OpOnSet"'] & \{\{f(a,a)\},\{f(b,b)\}\} \arrow[d, "Flat"] \\
\{\{\{f(a,a), f(a,b)\},\{f(b,a), f(b,b)\}\}\} \arrow[d, "Flat"'] & \{f(a,a), f(b,b)\} \\
\{f(a,a), f(a,b), f(b,a), f(b,b)\} &
\end{tikzcd}
$$

ConfStratStrict

The term $[l \to r](t)$ is reduced using the evaluation rule **Fire** only if $t \in \mathcal{T}(\mathcal{F})$.

ConfStrat

The term $[l \to r](t)$ is reduced using the evaluation rule **Fire** only if:

- the term l is linear,
- the term l weakly subsumes the term t ($\mathcal{FP}os(l) \subseteq \mathcal{FP}os(t)$),
- the term t contains no empty set and no set with more than one element,
- the term t contains no sub-term of the form $[u](v)$ where u is not a rewriting rule,
- for all sub-term $[u \to w](v)$ of t, u subsumes v,

  or

- $t \in \mathcal{T}(\mathcal{F})$ is a closed first order term.

ConfStratOper

The term $[l \to r](t)$ is reduced using the evaluation rule **Fire** only if $t \in \mathcal{T}(\mathcal{F})$ or:

- the term l is linear and
- the term l weakly subsumes the term t

  and

  - the rewrite rule $l \to r$ is conservative $(FV(l) \subseteq FV(r))$

    or

  - the term t contains no empty set
  - the term t contains no sub-term of the form $[u](v)$ where u is not a rewriting rule,
  - for all sub-term $[u \to w](v)$ of t, u subsumes v,

  and

  - the rewrite rule $l \to r$ is right linear

    or

  - the term t contains no set with more than one element.

When using one of the strategies *ConfStratStrict*, *ConfStrat* or *ConfStratOper*, the $\rho_\emptyset$-calculus in confluent.

About expressiveness of the ρ-calculus

Encoding the λ-calculus

$t ::= x \mid \{t\} \mid f(t, \ldots, t) \mid [t](t) \mid x \to t$

$$\textbf{FirePropagate} \quad [x \to r](t) \quad \mapsto\!\!\!\!\twoheadrightarrow \quad \{\{x/t\}r\}$$

$\varphi(\lambda x.t) = x \to t$

$\varphi((s\ t)) = [s](t)$

Given t and t' two λ-terms. Then, $t \longrightarrow_\beta t'$ iff $\varphi(t) \longrightarrow_\rho \{\varphi(t')\}$.

Proof: For any λ-term $(\lambda x.t\ u)$ that reduces to $\{x/u\}t$ the corresponding ρ-reduction is $[x \to t](u) \longrightarrow^{*}_{FirePropagate} \{\{x/u\}t\}$. $\square$

Encoding rewriting

$t ::= x \mid \{t\} \mid f(t,\ldots,t) \mid [t](l) \mid l \to l$

Given t and t' two terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{R}$ a first order term rewrite system.

If $t \xrightarrow{*}_{\mathcal{R}} t'$ then there exists a ρ-term u constructed using the rules in $\mathcal{R}$ such that $[u](t) \xrightarrow{*}_{\rho} \{t'\}$.

Proof: Let us consider a rewrite rule $(l \to r)$ that applies on the term t at position p and thus, $t[\theta l]_p \longrightarrow_{\mathcal{R}} t[\theta r]_p = t'$.

The ρ-term u to be applied in the ρ-calculus is $t[l \to r]_p$ and we get:

$$[t[l \to r]_p](t[\theta l]_p) \ \xrightarrow{*}_{\rho} \ \{t[\theta r]_p\}$$

$\square$

(Normalization) Strategies

Given a rewrite theory $\mathcal{R}$ does it exist a ρ-term $\xi_{\mathcal{R}}$ such that for any term u, if u reduces to the term v in the rewrite theory $\mathcal{R}$ then $[\xi_{\mathcal{R}}](u)$ ρ-reduces to $\{\ldots, v, \ldots\}$?

Given a rewrite theory $\mathcal{R}$ does it exist a ρ-term $\xi_{\mathcal{R}}$ such that for any term u, if u normalizes to the term v in the rewrite theory $\mathcal{R}$ then $[\xi_{\mathcal{R}}](u)$ ρ-normalizes to $\{\ldots, v, \ldots\}$?

Extending the calculus: with syntactic sugar

$$t ::= id \mid fail \mid t;t \mid dk(t,t)$$

Evaluation rules

Identity	id	$\mapsto\!\!\!\twoheadrightarrow$	$x \to x$
Fail	$fail$	$\mapsto\!\!\!\twoheadrightarrow$	$x \to \emptyset$
Compose	$[s_1; s_2](t)$	$\mapsto\!\!\!\twoheadrightarrow$	$[s_2]([s_1](t))$
DK	$[dk(s_1, s_2)](t)$	$\mapsto\!\!\!\twoheadrightarrow$	$\{[s_1](t), [s_2](t)\}$

The $first$ operator

First	$[first(s_1, \ldots, s_n)](t)$	$\mapsto\!\!\!\twoheadrightarrow$	$\langle [s_1](t), \ldots, [s_n](t) \rangle$
First_fail	$\langle \emptyset, t_1, \ldots, t_n \rangle$	$\mapsto\!\!\!\twoheadrightarrow$	$\langle t_1, \ldots, t_n \rangle$
First_success	$\langle t, t_1, \ldots, t_n \rangle$	$\mapsto\!\!\!\twoheadrightarrow$	$\{t\}$
			if $t \neq \emptyset$ *is closed and contains no redex*
First_single	$\langle \rangle$	$\mapsto\!\!\!\twoheadrightarrow$	$\{\}$

Term traversal

$$
\begin{array}{llcl}
\textbf{Tseq} & [\Phi(r)](f(u_1,\ldots,u_n)) & \twoheadrightarrow & \langle\{f([r](u_1),\ldots,u_n)\},\ldots,\{f(u_1,\ldots,[r](u_n))\}\rangle \\
\textbf{Tseq\_ct} & [\Phi(r)](c) & \twoheadrightarrow & \emptyset \\
\\
\textbf{Tpar} & [\Psi(r)](f(u_1,\ldots,u_n)) & \twoheadrightarrow & \{f([r](u_1),\ldots,[r](u_n))\} \\
\textbf{Tpar\_ct} & [\Psi(r)](c) & \twoheadrightarrow & \{c\}
\end{array}
$$

Fix point operators

$$\Theta = [A](A) \quad with \quad A = x \to (y \to [y]([[x](x)](y)))$$

Thus

$$[\Theta](G) \xrightarrow{*}_{\rho} \{[G]([\Theta](G))\}$$

Bottom-up application

$$Once_{bu}(r) \equiv [\Theta](H_{bu}(r))$$

where

$$H_{bu}(r) \equiv f \to (x \to [first(\Phi(f), r)](x))$$

Ex: $[Once_{bu}(a \to b)](f(a, g(a))) \Longrightarrow_{\rho} \{f(b, g(a))\}$

Repeat operator

$$repeat*(r) \equiv [\Theta](J(r))$$

where

$$J(r) \equiv f \to (x \to [first(r; f, id)](x))$$

Normalization strategies

$$im(r) \equiv repeat*(Once_{bu}(r))$$

$$om(r) \equiv repeat*(Once_{td}(r)).$$

Encoding Conditional Rewriting

The conditional rewrite rule:

$$l \to r \text{ if } c$$

is simply encoded by

$$l \to [\{\mathbf{T} \to r, \mathbf{F} \to \emptyset\}]([lmim](c))$$

Or even simpler

$$l \to [\mathbf{T} \to r]([lmim](c))$$

ELAN's semantics

Using rewriting logic [Meseguer 92]

Using ρ-calculus

The ELAN basic object: rewrite rules with where

Rules are labelled conditional rewrite rules with local variable affectations

$$
\begin{array}{lll}
[\mathsf{lab}] & l \Rightarrow & r(x,y) \\
& & \textbf{where } \; x := [S_1]u_1 \\
& & \textbf{if } \; c_1(x) \\
& & \textbf{where } \; y := [S_2]u_2 \\
& & \textbf{if } \; c_2(x,y)
\end{array}
$$

— lab is the rule label,

— l and r are the respective left and right-hand sides,

— c_1, c_2 are the conditions and

— $z := [S]u$ are local affectations, assigning to the local variable z one of the result of the strategy S applied to the term u.

Example of ELAN's rule

$$
\begin{array}{llll}
[\text{deriveSum}] & p_1 + p_2 & \Rightarrow & p_1' + p_2' \\
 & & & \mathsf{where}\ p_1' := (derive)p_1 \\
 & & & \mathsf{where}\ p_2' := (derive)p_2
\end{array}
$$

can be represented by the following two ρ-terms

$$p_1 + p_2 \to [derive](p_1) + [derive](p_2)$$

$$p_1 + p_2 \to [p_1' \to [p_2' \to p_1' + p_2']([derive](p_2))]([derive](p_1))$$

An expression of ELAN's rules in ρ-calculus

$$\begin{array}{lll} [\mathsf{lab}] & l \Rightarrow & r(x) \\ & & \textbf{where}\;\; x := [S_1]u_1 \\ & & \textbf{if}\;\; c_1(x) \end{array}$$

The rule is expressed as the ρ-term:

$$l \;\to\; [x \to [True \to r(x)]([lmim](c_1(x)))]([S_1](u_1))$$

With two `where if`

$$
\begin{array}{lll}
\text{[lab]} & l \Rightarrow & r(x) \\
& & \textbf{where } \; x := [S_1]u_1 \\
& & \textbf{if } \; c_1(x) \\
& & \textbf{where } \; y := [S_2]u_2 \\
& & \textbf{if } \; c_2(x,y)
\end{array}
$$

The rule is expressed as the ρ-term:

$$
\begin{array}{l}
l \to \quad [x \to [\quad True \to [\quad y \to [True \to r(x,y)]([lmim](c_2(x,y))) \\
\qquad\qquad\qquad\qquad\qquad\qquad]([S_2](u_2)) \\
\qquad\qquad\qquad]([lmim](c_1(x))) \\
\qquad]([S_1](u_1))
\end{array}
$$

To sum-up

- ρ_T-calculus is a simple, elegant and powerful new calculus
  — explicit rule application (and thus "matching power")
  — explicit handling of result sets
  — paramaterized by a matching theory T
- Clear distinction between the rewrite relation and the rewrite calculus.
- It allows uniform combination of first and higher-order computations
- ρ-calculus gives a simple semantics to ELAN programs

Back on elan

`http://www.loria.fr/ELAN`

Current release: V3.4

The ELAN team:
Peter Borovanský, Horatiu Cirstea, Hubert Dubois,
Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau,
Christophe Ringeissen, Marian Vittek.

Back on Fibonacci

$$\begin{array}{lcl} fib(0) & \rightarrow & 1 \\ fib(1) & \rightarrow & 1 \\ fib(n) & \rightarrow & fib(n-1) + fib(n-2) \end{array}$$

$$\boxed{fib(3)} \rightarrow \boxed{fib(2) + fib(1)}$$

$$fib(2) + \boxed{fib(1)} \rightarrow fib(2) + \boxed{1}$$

$$\boxed{fib(2)} + 1 \rightarrow \boxed{fib(1) + fib(0)} + 1$$

$$\boxed{fib(1)} + fib(0) + 1 \rightarrow \boxed{1} + fib(0) + 1$$

$$\ldots$$

In ELAN

```
module fib

import global int;
end

operators global
 fib(@)   : (int) int ;
end

rules for int
n : int ;
global
    [] fib(0) => 1 end
    [] fib(1) => 1 end
    [] fib(n) => fib(n - 1) + fib(n - 2) if n > 1 end
end
end
```

We need to specify how to start a computation

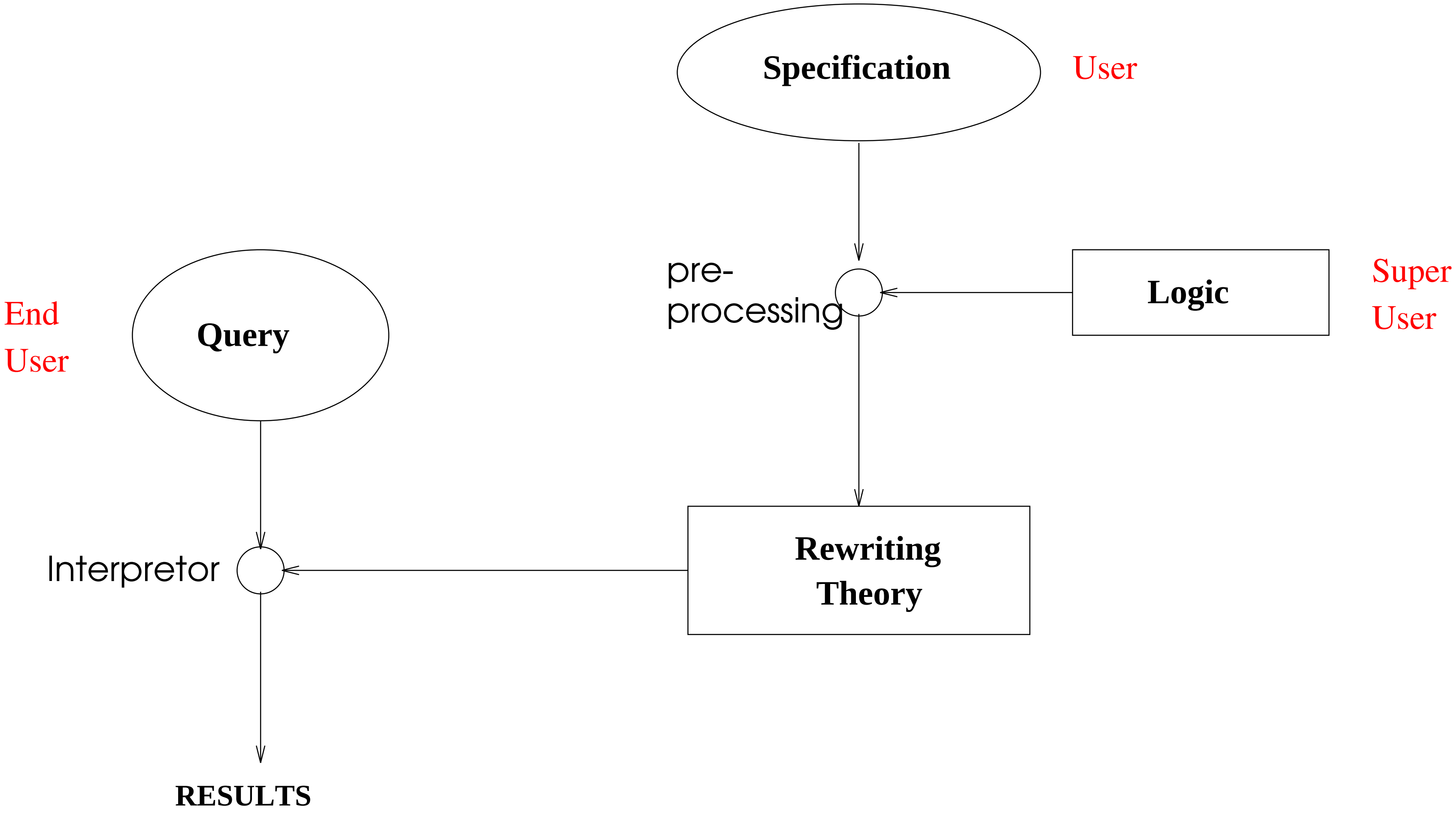
```
LPL fib description
  query    of sort int
  result   of sort int
  import   fib
  start with ()query
end
```

Then we can run it

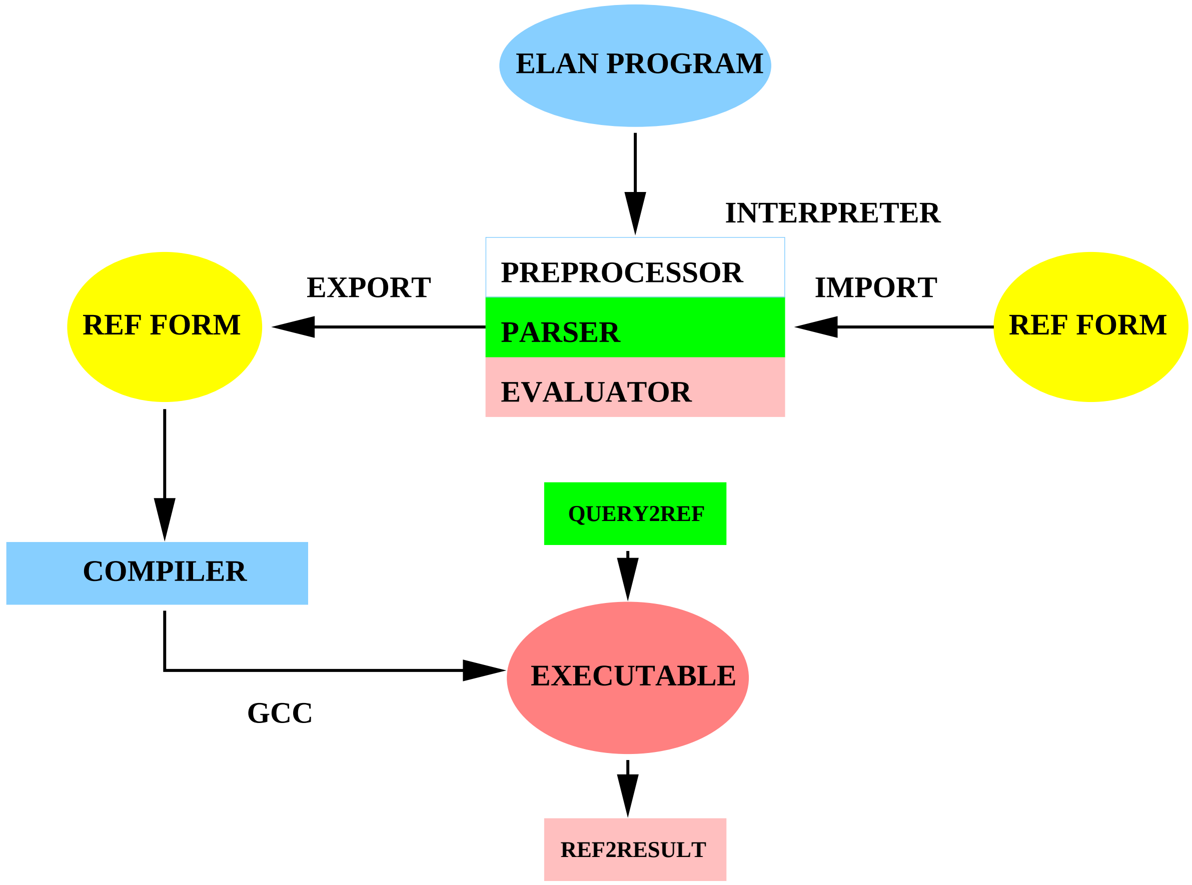
```
-hilbert 63-> elan fib

******** ELAN version 3.3 (08/12/98.16:18) ********

 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module fib
     handling module int
       handling module builtinInt
         handling module bool
         end of bool
       end of builtinInt
     end of int
   end of fib

enter query term finished by the key word 'end':
fib(4) end

[] start with term :
```

```
    fib(4)

[] result term:
    5

[] end

enter query term finished by the key word 'end':
```

ELAN: Logical Organization



ELAN: Environment



The language: Main principles

- Rules may be non-terminating and non-confluent
  but then their application has to be controlled by strategies
- Rules and strategies are first-class objects
- Application of rule or a strategy on a term may give
  0,
  1,
  or several results (could be infinite)

The language

Close to algebraic specification formalism
with additional specificities:

- Deductions and computations:
  - Rules for computations (confluent and terminating)
  - Rules for deductions (no confluence nor termination required)

- A strategy language to express control and derivation tree exploration.
  Non-determinism (sets of results) handled by backtracking.

- Generic language constructions handled by a pre-processor.

Programming styles

- Functional programs are naturally expressed
- Non-determistic computations are simply described
- Strategy programming for expressing the control

Modules

Functions, rules and strategies are defined in modules:
.eln file

Importations of already defined modules

that may be local or global

Predefined modules: bool,int,ident,list[X]...

```
module enum0
import  global int list[int] ; end
```

Signatures

- Sorts
- Functions with their ranks

```
sort int list[int];
end

operators global
  enum(@,@): (int int) list[int] ;
end
```

Rules

Conditional rewrite rules

```
rules for list[int]
    i,j,k,l: int ;
global
[] enum(i,j) => i.enum(i+1,j) if i<=j    end
[] enum(i,j) => nil if i>j               end
end
```

Top level

A .lgi file specifies the top module

and the sorts of the query and result.

```
LPL enum0 description
  query     of sort list[int]
  result    of sort list[int]
  import    enum0
  start with () query
end
```

```
******** ELAN version 3.3 (08/12/98.16:18) ********
 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module enum0
     handling module int
       handling module builtinInt
         handling module bool
         end of bool
       end of builtinInt
     end of int
     handling module list[int]
     end of list[int]
   end of enum0
enter query term finished by the key word 'end': enum(3,6) end
[] start with term :
    enum(3,6)
[] result term:
    3.4.5.6.nil
[] end
```

Introducing strategies

Instead of having `enum(3,6)`
as a list `3.4.5.6.nil`

is it possible to generate successively
`3, 4, 5, 6`?

Labelled rules

Rules are given a label.

```
rules for int
  i,j,k,l: int ;
global
[final] enum(i,j) => i end
[iter]  enum(i,j) => enum(i+1,j)
                     if i<j end
end
```

Strategy definition

A strategy operator defined by a rewrite rule on a strategy sort.

```
stratop global
  enumStrat1 : <int->int>          bs ;
end

strategies for int
implicit
[] enumStrat1 => iterate*(iter) ; final  end
end
```

```
******** ELAN version 3.3 (08/12/98.16:18) ********
 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module enum1
   ...
enter query term finished by the key word 'end':
enum(3,6) end
[] start with term :
    enum(3,6)
[] result term:
    3
[] result term:
    4
[] result term:
    5
[] result term:
    6
[] end
```

How is it computed?

```
[final] enum(i,j) => i
[iter]  enum(i,j) => enum(i+1,j)
                     if i<j

[] enumStrat1 => iterate*(iter) ; final
```

```
(enumStrat1) enum(3,6)
```

$iterate$	$iter$	$final$
$iterate0$	$enum(3,6)$	3
$iterate1$	$enum(4,6)$	4
$iterate2$	$enum(5,6)$	5
$iterate3$	$enum(6,6)$	6
$iterate4$	$fail$	$fail$

Elementary strategies

Built from a few built-in constructors:

- Labelled rule $[\ell]$
- Sequential composition $S_1; S_2$
- Iteration repeat*(S) or iterate*(S)

repeat*(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application.

iterate*(S) is similar to repeat*(S) but returns all intermediate results of repeated applications.

- Choice points

dk$(S_1, \ldots, S_n)$: all results of all S_i

dc$(S_1, \ldots, S_n)$: all results of one S_i

first$(S_1, \ldots, S_n)$: all results of the first S_i

- Cut points

dc one$(S_1, \ldots, S_n)$: one result of one S_i

first one$(S_1, \ldots, S_n)$: one result of the first S_i

- Failure **fail** and Identity **id**

Labelled rules

```
module tree
import global int ; end
sort int tree ; end
operators  global
        @               : (int) tree;
        leaf(@)         : (int) tree;
        node(@,@,@)     : (tree int tree) tree;
end
rules for tree
        l, r: tree;  n: int;
global
[Left]     node(l,n,r)    => l             end
[Right]    node(l,n,r)    => r             end
[L_value]  leaf(n)        => n             end
end
```

Strategies

```
stratop
  global
        allleaves  : <tree -> tree>        bs;
end

strategies for tree
implicit
  [] allleaves  => iterate*(dk(Left,Right));L_value    end
end
end
```

```
******** ELAN version 3.3 (08/12/98.16:18) ********
 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module tree
   ...
enter query term finished by the key word 'end':
node(node(leaf(1),2,leaf(3)),4,node(leaf(5),6,leaf(7))) end
[] start with term :
    node(node(leaf(1),2,leaf(3)),4,node(leaf(5),6,leaf(7)))
[] result term:
    1
[] result term:
    3
[] result term:
    5
[] result term:
    7
[] end
```

Rules with local variables and patterns

Labelled rules are always applied at top position.
Local variables used to direct strategy application

```
rules for poly
  p1, p2, p3, p4, p5 : poly; x variable;
global
  []  deriv(p1*p2,x)   => p5
        where p3:=(simplify) deriv(p1,x)*p2
        where p4:=(simplify) p1*deriv(p2,x)
        where p5:=(simplify) p3+p4 end
end
strategies for poly
  [] simplify => repeat*(first one(expand)) ;
                 repeat*(first one(factorize))
end
```

Rules with local patterns

Local variables generalise to patterns (a term with variables)

```
[] move(S) => C(x,y)
      where < x , y > := position(S)
      if x=y
```

General form of rules

$$[\ell] \quad l \rightarrow r \ \textbf{where}\ p_1 := (S_1)c_1 \ldots \ \textbf{where}\ p_n := (S_n)c_n$$

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\mathcal{V}ar(()p_i) \cap (\mathcal{V}ar(()l) \cup \mathcal{V}ar(()p_1) \cup \cdots \cup \mathcal{V}ar(()p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(()r) \subseteq \mathcal{V}ar(()l) \cup \mathcal{V}ar(()p_1) \cup \cdots \cup \mathcal{V}ar(()p_n)$ and
- $\mathcal{V}ar(()c_i) \subseteq \mathcal{V}ar(()l) \cup \mathcal{V}ar(()p_1) \cup \cdots \cup \mathcal{V}ar(()p_{i-1})$.

where $true := c$ is written **if** c.

p_i often reduced to a variable x.

S_i may be the identity strategy, written ().

Application of a rule

$$[\ell] \quad l \rightarrow r \ \textbf{where}\ p_1 := (S_1)c_1 \ldots \ \textbf{where}\ p_n := (S_n)c_n$$

To apply the rule to a subject t, the matching substitution from l to t ($l\sigma = t$) is successively composed with each matching μ_i from p_i to $(S_i)c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$.

Evaluation of $(S)t$:
1. t is normalized using the unlabelled rules
2. One tries to apply a labelled rule according to the strategy S.

Choice points are set when there are several results.
Backtrack occurs when the set of results in empty.

Rules with Associative Commutative functions

If l or p contains AC function symbols, AC matching is called and can return several solutions.

One may want to get all solutions of an AC matching problem and build all possible results of rewriting with these different matching substitutions.

When the rule contains a sequence of matching conditions, failing to satisfy the i-th condition causes a backtracking to the previous one.

```
operators global
  @ U @ :(set set) set (AC) ;
  emptySet : set ;
  (@): (int) set ;
  extract(@) : (set) int ;
end
rules for int
   i: int ; s: set ;
global
[Rule] extract( (i) U s )  => i end
end
```

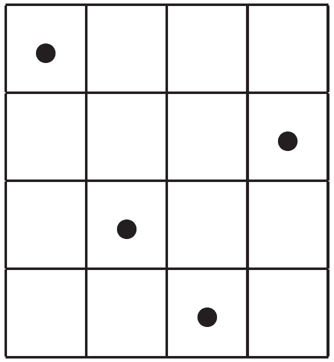
Application of the rule `[Rule]` to
`extract(emptySet U (1) U (2) U (3) U (4) U (5))`
returns one of `1,2,3,4,5`.
Application of the strategy `dk(Rule)` returns all these results successively.

```
module enumAC3
import  global int ; end
sort set ;  end

operators global
  @ U @ :(set set) set (AC) ;
  emptySet : set ;
  (@): (int) set ;
  enum(@,@): (int int) set ;
  extract(@) : (set) int ;
end

stratop global
  enumStratAC : <int -> int>         bs ;
end
```

```
rules for set
  i,j: int ;
global
[] enum(i,j) => emptySet if i>j       end
[] enum(i,j) => (i) U enum(i+1,j)
                      if i <= j           end
end
rules for int
  i: int ; s: set ;
global
[Rule] extract( (i) U s )  => i end
end
strategies for int
implicit
[] enumStratAC => dk(Rule) end
end
```

```
******** ELAN version 3.3 (08/12/98.16:18) ********
 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module enumAC3
   ...
enter query term finished by the key word 'end':
enum(3,6) end
[] start with term :
    extract(enum(3,6))
[] result term:
    6
[] result term:
    5
[] result term:
    4
[] result term:
    3
[] end
```

N queens



is represented by the list (3,1,0,2)

Module nqueensAC

```
operators
 global
    queensAC(@) : (int) list[int] ;
 local
    queens(@,@) : (set list[int]) list[int] ;
    ok(@,@,@)   : ( int int list[int] ) bool;
    @ U @       : (set set) set (AC);
    (@)         : (int) set ;
    [@ U @]     : (int set) set ;
    Set(@)      : (int) set ;
    Empty       : set ;
end
```

```
stratop
  global
    queens     : <list[int] -> list[int]>   bs;
  local
    extractPos : <set -> set>               bs;
end
```

```
rules for set
  S: set ;
  i: int ;
global
  [] Set(0) => Empty U (0) end
  [] Set(i) => Set(i-1) U (i) end
  [extractrule] (i) U S => [i U S] end
end

strategies for set
implicit
  [] extractPos => dk(extractrule) end
end
```

```
rules for bool
  p,d,diff : int;
  l : list[int];
global
  [] ok(diff,d,nil) => true end
  [] ok(diff,d,p.l) => false         if d-p == diff  end
  [] ok(diff,d,p.l) => false         if p-d == diff  end
  [] ok(diff,d,p.l) => ok(diff+1,d,l) end
end
```

```
rules for list[int]
  n:int;
global
  [] queensAC(n) => queens(Set(n-1),nil)

rules for list[int]
  p1: int;  s,s1:set ;  l,l1: list[int];
local
  [final] queens(Empty,l) => l     end
  [queensrule]  queens(s,l) => queens(s1,p1.l)
                  where (set) [p1 U s1] :=(extractPos) s
                  if ok(1,p1,l)

strategies for list[int]
implicit
  [] queens => repeat*(dk(queensrule)); dc(final)
```

Execution

```
Query term: queensAC(4) end
1.3.0.2.nil
2.0.3.1.nil
2 solutions

Query term: queensAC(8) end
...
3.1.6.2.5.7.4.0.end
92 solutions
```

Execution with the compiler

On a Pentium II 450mhz, linux, with `-optimiseChoicePoint`.

```
queens(8) -- 92 results
rewrite_step = 106049
0.150u 0.020s

queens(10) -- 724 results
rewrite_step = 2608909
1.850u 0.230s

queens(12) --  14200 results
rewrite_step = 82711249
51.230u 2.090s

queens(14) --- 365596 results
rewrite_step = 3307192557
1892.910u 60.260s
```

Defined Strategies

Strategy operators with arguments (parameters).

$$\mathbf{map} \;:\; (\langle s\rangle)\langle list[s]\rangle$$

$$\mathbf{map}(S) \Rightarrow \mathbf{dc}(\mathbf{nil}, S \cdot \mathbf{map}(S))$$

where $S : \langle s\rangle$ and $s \in \mathcal{S}$.

equivalent to

$$[\mathbf{map}(S)]nil \Rightarrow nil$$

$$[\mathbf{map}(S)]x \cdot l \Rightarrow [S]x \cdot [\mathbf{map}(S)]l$$

```
module map[X]

import global
  strat[X] strat[list[X]] X list[X]; end
stratop global
  map(@)  : (<X->X>) <list[X]->list[X]>;
  mac(@)  : (<X->X>) <list[X]->list[X]>;
end

rules for X
 x : X;  n,m : int;
global
  [mul2] x => x*2                    end
end
```

```
strategies for list[X]
 s : <X->X>;  h : X;  h1 : X;  t : list[X];  t1 : list[X];
explicit
[.] [mac(s)]nil => nil                    end
[.] [mac(s)]h.t => h1.t1
       where h1:=[s]h
       where t1:=[mac(s)]t                end
implicit
[.] map(s) => dc(nil,cons(s,map(s)))    end
end
end
```

```
******** ELAN version 3.3 (08/12/98.16:18) ********
 (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1998
   handling module int
   ....
   end of map[int]

enter query term finished by the key word 'end':
    3.4.5.6.nil
[] start with term :
    3.4.5.6.nil
[] result term:
    6.8.10.12.nil
[] end
```

Strategy terms

- Elementary strategies: built from strategy constructors, applied with $(\_)\_$ (built-in).
- Defined strategies: built from user's strategy operators, applied with $[\_]\_$ (meta-interpreter).

Strategy application

- Evaluation of $(S)t$:
  1.t is normalized using the unlabelled rules
  2.One tries to apply a labelled rule according to the strategy S.

- Evaluation of $[S]t$:
  1.S and t are normalized using the unlabelled rules
  2.rules defining $[\_]\_$ are applied according to a strategy `eval` (defined in the meta-interpreter).

Rules on strategy terms

- When the rule is labelled by `[.]`, it is used by the strategy meta-interpreter (ELAN program) according to its strategy `eval`.
- When the rule is labelled by $[\ell]$, it is used by the ELAN interpreter as any other labelled rule governed by a strategy.
- If the rule is unlabelled, the leftmost innermost predefined strategy is applied.

Deductions modulo Computations

– **Labelled rules** whose application is fully controled by the user elementary or defined strategies and,

– **Unlabelled rules** applied using a built-in leftmost innermost strategy (confluent and terminating).

Deductions, performed by labelled rules under strategies
versus
Computations, performed by unlabelled rules.

Some Applications developed in ELAN

- Theorem provers (KB completion, B provers)
- Automata computations and approximations
- COLETTE: Constraint Satisfaction Problems
- Unification algorithms
- CLP, Combination of solvers
- Production rules (planification)

Why is this kind of programming language useful?

- Human thinking is very often of transitional kind
- The underlined semantics is well defined and mathematicaly founded
  - uniform integration of first and higher-order features
  - rewriting calculus
  - rewriting logic
- It allows us to set up safe programming environment ...

Example: Is this program safe?

```
module quotientRemainder

sort nat pairNat ;
end

operators global
 0 : nat ;
 s(@)   : (nat) nat ;
 [@,@] : (nat nat) pairNat ;
 s(@) : (pairNat) pairNat ;
 q(@,@) : (nat nat) pairNat ;
 qq(@,@,@,@) : (nat nat nat nat) pairNat ;
// First argument of qq is number to be divided minus 1 (when called)
// Second argument is the divider minus 1 (when called)
// Third argument is the divider (just to remember)
// Last argument computes the potential remainder
end
```

```
rules for pairNat
x, y, z, r : nat ;
global
    [] q(0,s(y))  => [0,0]  end
    [] q(s(x),s(y))  => qq(x,y,s(y),s(0))  end
    [] qq(x,0,z,r) => s(q(x,z)) end
    [] s([x,y]) => [s(x),y] end
    [] qq(0,s(y),z,r) => [0,r] end
    [] qq(s(x),s(y),z,r) => qq(x,y,z,s(r)) end
end
```

Proof assertion

```
Theorem QR in quotientRemainder
forall a: nat, b: nat,  a = b * pr1(q(a,b)) + pr2(q(a,b))
// where pr1 and pr2 are the respective projections
and 0 =< pr2(q(a,b)) < b
end
```

Then not only one can apply the rules in order to compute the quotient and remainder of any two naturals, but on can also prove that

$$\texttt{quotientReste} \models \forall a : nat, \forall b : nat, \quad \begin{array}{l} a = b * pr1(q(a,b)) + pr2(q(a,b)) \\ \wedge \\ 0 \leq pr2(q(a,b)) < b \end{array}$$

Not yet available!

The full integration of the specification and proof levels is not yet available but practical examples are

– already reachable and

– quite appealing with respect to existing technologies

For example the NSPK verification:

Encoding the Needham-Schroeder public-key protocol in ELAN

- user-definable number of initiators and responders
- user-definable size of the network
- the strategies guiding the rewrite rules describe a form of model-checking in which all the possible behaviors are explored

Data structures

- The agent

```
@ + @ + @  : ( AgentId SWC Nonce ) Agent;
@          : ( Agent ) listAgent;
@ || @     : ( listAgent  listAgent ) listAgent (AC);
```

- The messages

```
@-->@:@[@,@,@] : ( AgentId AgentId Key Nonce Nonce Address ) message;
@         : ( message ) network;
@ & @     : ( network  network ) network (AC);
```

- The intruders

```
@ # @ # @ : ( AgentId listNonce network ) intruder;
```

- The global state

```
@ <> @ <> @ <> @ : ( listAgent listAgent intruder network ) state;
```

Rewrite rules for the agents

- initiator starts the communication with a responder

```
[initiator-1]
   x+SLEEP+resp  || E <> D <> w#l#ll <> ls                              =>
   x+WAIT+N(x,y) || E <> D <> w#l#ll <> x-->y:K(y)[N(x,y),DN,A(x)] & ls
                 where (Agent)y+std+init :=(extAgent) elemIA(D || w+SLEEP+DN)
end
```

- responder reads the message and sends the acknowledgement

```
[responder-1]
   E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)] & ls =>
   E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)] & ls
end
```

Rewrite rules for the agents

- initiator receives the acknowledgement and checks its validity

```
[initiator-2]
 x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls =>
 S
  choose
    try
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls
    try
      if x!=n1 or v!=n3
      where S:=() ERROR
  end
end
```

Rewrite rules for the intruder

- the intruder intercepts all the messages in the network but the messages generated by itself and stores or decrypts them.

```
[intruder-1]
   E <> D <> w#l#ll <> z-->x:K(w)[N(n1,n3),N(n2,n4),A(v)] & ls  =>
   E <> D <> w#N(n1,n3) | N(n2,n4) | l#ll <> ls
                  if w!=z
end
```

- the nonces obtained previously by the intruder are used in order to generate fake messages that are sent to all the agents.

```
[intruder-4]
   E <> D <> w # resp | l # ll <> ls                          =>
   E <> D <> w # l # ll <> w-->y:K(y)[resp,DN,A(xadd)] & ls
                 where (Agent)y+std+dn :=(extAgent) elemIA(D || E)
                 where (Agent)xadd+std1+dn1 :=(extAgent) elemIA(D || E)
end
```

The invariants

- authenticity of the responder: if an initiator `x` committed with a responder `y`, then `y` has really been involved in the protocol.

```
[attack-1] x+COMMIT+N(x,y) || E <> D <> I <> ls =>
           ATTACK
                if y!=i
                if not(existAgent(y+WAIT+N(y,x),D)) and
                   not(existAgent(y+COMMIT+N(y,x),D))
end
```

- authenticity of the initiator: if a responder `y` committed with an initiator `x` then the initiator have committed as well with `y`.

```
[attack-2] E <> y+COMMIT+N(y,x) || D <> I <> ls =>
           ATTACK
                if x!=i
                if not(existAgent(x+COMMIT+N(x,y),E))
end
```

The strategy

We apply repeatedly all the rewrite rules in any order and in all the possible ways until one of the attack rules can be applied.

```
[]attStrat  =>  repeat*(
                   dk(
                      attack-1, attack-2,
                      intruder-1, intruder-2, intruder-3, intruder-4,
                      initiator-1, initiator-2, responder-1, responder-2)
                );
                attackFound
end
```

where

```
[attackFound]   ATTACK   =>   ATTACK             end
```

The attack

$$
\begin{array}{lll}
\text{I.1.} & A \to I & : \{N_A, A\}_{K(I)} \\
\text{II.1.} & I(A) \to B & : \{N_A, A\}_{K(B)} \\
\text{II.2.} & B \to I(A) & : \{N_A, N_B\}_{K(A)} \\
\text{I.2.} & I \to A & : \{N_A, N_B\}_{K(A)} \\
\text{I.3.} & A \to I & : \{N_B\}_{K(I)} \\
\text{II.3.} & I(A) \to B & : \{N_B\}_{K(B)}
\end{array}
$$

A second strategy

```
[]attStrat  =>  repeat*(
                    dk(
                       attack-1, attack-2,
                       initiator-1, initiator-2, responder-1, responder-2,
                       intruder-1, intruder-2, intruder-3, intruder-4)
                );
                attackFound
end
```

I.1.	$A \to I$	$: \{N_A, A\}_{K(I)}$
II.1.	$I(A) \to B$	$: \{N_A, A\}_{K(B)}$
I+II.2.	$B \to A$	$: \{N_A, N_B\}_{K(A)}$
I.3.	$A \to I$	$: \{N_B\}_{K(I)}$
II.3.	$I(A) \to B$	$: \{N_B\}_{K(B)}$

The corrected protocol

1. $A \to B$: $\{N_A, A\}_{K(B)}$
2. $B \to A$: $\{N_A, N_B, B\}_{K(A)}$
3. $A \to B$: $\{N_B\}_{K(B)}$

Modified rule: `initiator-2`

```
[initiator-2] x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls  =>
S
  choose
    try
      if v==z               // expected responder
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls
    try
      if v!=z               // not expected responder
      if x!=n1 or v!=n3
      where S:=() ERROR
  end                                                                          end
```

Optimizations of the current implementation

- each message is given a type according to its relevant fields (nonce and address),
- initiator and responder listen only to messages coming from the intruder,
- the intruder sends fake messages only to those agents that are able to handle it

Comparison with Murφ

ini	res	net	Murφ	ELAN simple	ELAN(Murφ like)	ELAN optimized
1	1	1	2578 rules 0.56s	7442 rules 0.167s	4308 rules 0.125s	711 rules 0.064s
1	1	2	136273 rules 18.40s	453514 rules 7.007s	333552 rules 4.980s	6700 rules 0.162s
2	1	1	25701 rules 6.81s	575101 rules 8.571s	257087 rules 3.946s	26011 rules 0.483s
2	2	1	557430 rules 303.46s	118214389 rules 1658.296s	22985807 rules 333.096s	753785 rules 12.392s

Protocol verification: some conclusions

- computational systems can be used as a logical framework for representing the Needham-Schroeder public-key protocol,
- this approach can be easily extended to other authentication protocols (TMN),
- the strategy used for guiding the application of the rewrite rules is important when an attack on the protocol exists,
- the behavior of intruders can be easily modified by changing the corresponding rules,
- the state space is finite and thus, some other methods should be used in order to show that properties proved for the finite model can be lifted to an unbounded model,

ELAN Version 3.4 (Mars 2000) `http://www.loria.fr/ELAN`

- An interpreter of the language,
- A very efficient compiler (runs upto 15 million rules per second),
- User's manual and library documentation,
- Examples; from tiny to very large ones.

ELAN is written in C++ and the current compiler is written in JAVA. The system runs on Suns and Digitals under Unix as well as on PCs under Linux.

We maintain `elan-users-request@loria.fr`, a low-volume moderated mailing list for announcements and discussion regarding ELAN.

Some ongoing works on ELAN and the ρ-calculus

- Cooperation CoQ – ELAN
- Constraints as first class objects
- ELAN and production rule systems
- typed versions of the ρ-calculus
- ρ-calculus with explicit substitutions

Next: adventures in Deduction Modulo