Proving self-stabilization with a proof assistant

Pierre Courtieu

***Abstract*—We present a formalization of a proof of self-stabilization in the Coq proof assistant. Coq is a program allowing to define mathematical objects and properties, and to make proofs on them in a certified way. We use it to formalize a generic proof of stabilization for algorithms running on a linear net or a ring[1]. In this method net configurations are considered as words and the algorithm as a rewriting system on words.**

***Index Terms*—Self-stabilizing algorithm, proof assistant, rewriting, regular expression.**

I. Introduction

We present a development on self-stabilization in the Coq proof assistant ([2]). We use the description of a generic method for proving self-stabilization of a certain class of algorithms by Beauquier, Bérard, Fribourg and Magniette [1]. This method, that we will call *TRC (top reduction chains)* in the following, has two parts:

1. The first part is generic and must be done and proved once and for all. The formalization of this part is the man subject of this paper;
2. The second part is to be done for each algorithm considered specifically, which means that a development in Coq, using the generic part (packed in a Coq library), must be done for each algorithm. We will show an example of this part at the end of this paper. (Semi-) automation of this second part is shortly discussed in the conclusion.

Proofs of stabilization are often difficult and need generally a deep understanding of the algorithm considered. Generic proof methods are in that case very helpful, as they ease a difficult part of the work. Such methods require some preconditions in order to be applied.

The application of a method and the verification of its preconditions are therefore critical to ensure the correctness of the whole proof. This is where proof assistants can help, particularly tools which like Coq provide a certified *proof-checker*.

The present work shall be viewed in two ways:

- To our knowledge it is the first computer assisted formalization of a proof method for stabilization,
- It is also the first step toward a dedicated environment for making certified proofs of self-stabilization.

A. *The proof assistant* Coq

Coq [2] is a program that allows to make mathematical proofs based on the type theory. The main principle of this program is based on the *Curry-Howard isomorphism* [3] that we describe very briefly here.

A.1 The Curry-Howard isomorphism

This isomorphism, initially stated between the typing rules of λ-calculus and inference rules of natural deduction, states that formal proofs à la natural deduction and terms of a typed calculus are isomorphic. More precisely, when we state that the term t of the simply typed λ-calculus is of type T, it is equivalent to say that t is a proof of T in natural deduction. We illustrate this principle by the following comparison between some rules of the two formal systems:

Natural deduction	Simply typed λ-calculus
$\dfrac{A \in \Gamma}{\Gamma \vdash A}$	$\dfrac{(x:A) \in \Gamma}{\Gamma \vdash x : A}$
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash u : A \quad \Gamma \vdash t : A \rightarrow B}{\Gamma \vdash (t\ u) : B}$
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma :: (x:A) \vdash t : B}{\Gamma \vdash [x:A]t : A \rightarrow B}$

We see here that types correspond to properties, and terms to formal proofs. *The process of building a proof of a certain property P corresponds to building a term of type P.* For a full review on the Curry-Howard isomorphism, see [4].

A.2 Coq

The program Coq is a tool that allows the user to build proofs. Internally, it builds λ-terms in the Curry-Howard spirit. The calculus used is not the simply typed λ-calculus, but the *Calculus of Inductive Constructions*(CIC), first described in [5]. It is a powerful λ-calculus featuring polymorphism, dependent types and *inductive types*. Inductive types allow to define new types by enumerating there *constructors* like in functional languages. For example, the type *nat* in Coq is defined as follows (all Coq definitions will be written in `verbatim` style):

```
Inductive nat:Set :=
 | O: nat
 | S: nat -> nat.
```

Induction principles are automatically generated for inductive types. By combining dependent and inductive types, we can define *inductive properties*, like for example the relation *le*($\leq$) below:

```
Inductive le: nat -> nat -> Prop :=
 | le_n: (n:nat)(le n n)
 | le_S: (n,m:nat)
        (le n m) -> (le n (S m)).
```

Where `(x:nat)P` must be read: $\forall x$: `nat.P`[1].

If we can find a term t of type `(le O (S O))`, then it is a proof that $0 \leq 1$. Such a term exists, for example `(le_S O O (le_n O))`, so we conclude that $0 \leq 1$ is provable. Of course we will not be able to find a term of type `(le (S O) O)` in a consistent context.

Coq allows the user to:

1. Define mathematical objects, i.e.

Laboratoire de recherche en informatique
Université Paris XI, 91405 Orsay cedex, France
email: courtieu@lri.fr

[1] Similarly in the following `[x:T]P` must be read λx:T.P

- Basic objects like integers, lists etc;
- Properties on these objects, like `le`;

2. Build proofs of the properties. In order to achieve this Coq has a language of *tactics*, allowing to build proofs interactively.
3. Verify built proofs (i.e. terms), by inferring there type.

Objects, types and proofs are all terms of CIC, which means in particular that when proving properties of programs, specifications and proofs are done in the same language. Actually, even programs can be expressed in the same language, and an extraction mechanism is available that produces certified ML code from Coq definitions. We mean by certified code that the properties proved on a Coq program are also verified by the ML program. In other words proof assistants are also environments for certified programming.

A.3 The section mechanism of Coq

We will make a frequent use of a practical feature of Coq: the section mechanism. Starting a *section* allows to make temporary declarations that will be abstracted at the end of the section. This corresponds to the mathematical use of the sentence "let X be some Y in this section". We illustrate this feature with the definition of the properties `Rstar` (transitive/reflexive closure of a relation), `Rplus` (transitive closure of a relation) and `Runion` (union of two relations), that we will use later.

First we start a section.

```
Section star_section.
```

Then we declare (temporarily) a relation `R` over a type `A`.

```
(* Let A be a type *)
Variable A:Set.
(* Let R and Q be two relations on A *)
Variable R,Q: A -> A -> Prop.
```

Then we define `Rstar`, `Rplus` and `Runion`:

```
Inductive Rstar: A -> A -> Prop :=
 | star_eq : (x:A)(Rstar x x)
 | star_tran:(x,y,z:A)
    (Rstar x y) -> (R y z) -> (Rstar x z).

Inductive Rplus: A -> A -> Prop :=
 |  Rplus_1step: (x,y:A)
              (R x y)->(Rplus R x y)
 | Rplus_trans1: (x,y,z:A)
   (R x y)->(Rplus R y z) -> (Rplus R x z).

Definition Runion := [x,y:A] (Q x y)\/(R x y).
```

Finally we close the section, and ask Coq the definition of `Rstar`.

```
End star_section.
Print Rstar.
Inductive Rstar
[A : Set; R : A->A->Prop]  : A->A->Prop :=
 star_eq : (x:A)(star A R x x)
| star_tran : (x,y,z:A)
  (Rstar A R x y)->(R y z)->(Rstar A R x z)
```

We see that the temporary objects used in the definition of `Rstar` are now parameters of the definition of `Rstar`.

Moreover, the ability in Coq to have implicit arguments allows to use the notation `(Rstar R x y)` instead of `(Rstar A R x y)` in the following. These three properties will be used in the following.

A.4 Termination of reductions in Coq

Reductions will be represented in Coq by relations, i.e. terms of type `A -> A -> Prop`, where `A` is the type of elements. The type `(R x y)` stands for the property $x \longrightarrow_R y$.

We give now an example of inductive property, that is close to the notion of *convergence* of a self-stabilizing algorithm. This inductive predicate `Acc` (for accessibility) is such that for a reduction relation `R` on a type `T` and a term `x`, if `(Acc R x)` is provable, then all reductions by `R` from `x` terminate[2]:

```
Inductive Acc  : (A -> A -> Prop) A->Prop :=
 | Acc_intro :  (R:A -> A -> Prop)(x:A)
    ((y:A)(R x y)->(Acc R y))->(Acc R x).
```

Intuitively, we see that:

- A term `t` that has no successor by `R` will verify `(Acc R t)` because
  `(y:A)(R x y)->(Acc R y)`
  (read: $\forall y.x \rightarrow_R y \Rightarrow (\texttt{Acc}\, \mathrm{R}\, y)$)
  is trivially provable.
- If a term has n successors by `R`, then it is accessible if and only if all the n successors are accessible.

B. *Self-stabilization*

Self-stabilizing algorithms[6] are distributed algorithms having the following property:

Definition I-B.1 (Self-stabilization 1) Let A be a distributed algorithm executing on a net Δ, let $\mathcal{L}$ be the set of legitimate configurations of Δ. A is said self-stabilizing for $\mathcal{L}$ iff, for all executions of A from any initial configuration the net reaches a configuration of $\mathcal{L}$ after a finite number of steps and never steps out of $\mathcal{L}$ anymore.

An almost equivalent definition is the following:

Definition I-B.2 (self-stabilization 2) Let A be a distributed algorithm executing on a net Δ, of which $\mathcal{L}$ is the set of legitimate configurations. A is said self-stabilizing for $\mathcal{L}$ iff:

1. Any execution of A from any initial configuration reaches $\mathcal{L}$ in a finite number of steps(*convergence*),
2. Any transition made by A from a configuration of $\mathcal{L}$ reaches a configuration of $\mathcal{L}$(*closure*).

These two definitions are closely related: a self-stabilizing algorithm for a set $\mathcal{L}$ in the sense of the first definition is self-stabilizing for a subset of $\mathcal{L}$ in the sense of the second definition. We will use the second definition in the following.

The main advantage of self-stabilization is the *fault tolerance*: after a fault of a node leaving the whole distributed system in an inconsistent state, provided that no other fault arise in a sufficiently long period, the system will converge in a consistent state again.

Let us notice that both definitions imply that the algorithm can't stop outside of $\mathcal{L}$, this is the *no-deadlock* condition.

II. Linear networks

The distributed algorithms considered in this paper are algorithms executing on linear nets, i.e. nets on which each node has at most two neighbors, the characteristics of this topology are the following:

[2] For the sake of clarity, the definition of `Acc`, is not exactly the one used in Coq: the arguments are switched in order to follow the intuition that `R a b` means a$\longrightarrow_R$B.

- Configurations can be represented by words on the alphabet of states of nodes. In the case of a ring, we will consider a particular node in order to designate the first letter of the word.
- Distributed algorithms can be defined as rewriting systems on words (i.e. configurations).
- As each node can communicate only with its two neighbors in the net, left members of the rewriting system will be at most three letters long. Since the topology of the net does not change during the execution of the algorithm, the size of words is not modified by the rewriting system. Therefore this system will have rules of the form:

$$\begin{array}{rl} & abc \longrightarrow ab'c \\ or: & ab \longrightarrow ab' \\ or: & ab \longrightarrow a'b \end{array}$$

  where a, b and c are letters representing possible states of the nodes.

In this paper we focus on algorithms having this form, however the first part of the definitions we give (until section IV-A) are completely generic and make no assumption on the topology of the network.

We give now the example of an algorithm that the TRC.

III. An Example: Ghosh's Algorithm

The Ghosh's 4 states algorithm[7] is a mutual exclusion algorithm. The system considered is a linear net of N machines, numbered from 0 to $N-1$. Each node has four possible states: $\{0,1,2,3\}$, except the node 0 (the *bottom* machine) that belongs to $\{1,3\}$ and node $N-1$ (the *top* machine) that belongs to $\{0,2\}$.

Remark III-.3: The Ghosh's algorithm, as the other algorithms that we consider, distinguishes a particular node of the net, and may apply certain particular rules on it. This particular node is the *top* node, i.e. the last letter of the word representing the net.

Remark III-.4: For clarity we may sometime represent the beginning and end of a configuration by a particular additional letter $\#$. For example the words $\#000\#$, $\#101010\#$ represent respectively configurations of nets of size 3 and 6. Using this additional letter, left and right members of rewrite rules will always be of size 3.

A. The set $\mathcal{L}$

The set $\mathcal{L}$ of legitimate configurations is defined by the following regular expression:

$$\mathcal{L} = \#\{1,3\}^+\{0,2\}^+\#$$

B. The algorithm

The algorithm is defined by the following rewriting system over configurations:

$$\begin{array}{lll} M_1: & \#X(q+1)qY\# & \longrightarrow \#X(q+1)(q+1)Y\# \\ & & \longrightarrow \text{if } Y \neq \varepsilon \\ M_2: & \#Xq(q+1)Y\# & \longrightarrow \#X(q+1)(q+1)Y\# \\ & & \longrightarrow \text{if } X \neq \varepsilon \\ T_1: & \#X32\# & \longrightarrow \#X30\# \\ T_1: & \#X10\# & \longrightarrow \#X12\# \\ B_1: & \#12Y\# & \longrightarrow \#32Y\# \\ B_1: & \#30Y\# & \longrightarrow \#10Y\# \end{array}$$

where $+$ is the sum modulo 4 and $q \in \{0,1,2,3\}$.

We can distinguish three kinds of rule:

- The *Bottom* rules (B_i) which modify the bottom letter,
- The *Top* rules (T_i) which modify the top letter.
- The *Middle* (M_i) which modify other letters.

Here we see for example that the rule M_1 is simultaneously a middle and a top rule (when $X = \varepsilon$), it is always possible to reformulate the rules in order to make T, B and M disjoint.

In this paper, we focus on algorithms having this presentation and we will formalize the recent method for proving self-stabilization for this kind of algorithms in [1]. We will precise the additional hypothesis necessary to this method during the development.

IV. Definition of self-stabilization in Coq

In this section we define in Coq the notions of self-stabilization and distributed algorithm over a linear network. These definition are general and is independent of the method of proof of self-stabilization of [1].

A. General definition of self-stabilization

This part is independent of the topology of the system considered. As we will see the type of configurations is a parameter of the following definition, and no assumption is made on it.

We first open a section and declare as temporary objects a type `wrd` representing configurations, a relation `R` representing the algorithm, and a predicate `L` representing the set $\mathcal{L}$ of legitimate configurations. These three notions will be parameters of the following definitions of the section.

```
Section selfstab.

(* Configurations *)
Variable wrd:Set.

(* The algorithm *)
Variable R: wrd -> wrd -> Prop.

(* Legitimate configurations *)
Variable L: wrd  -> Prop.
```

Then we define the property of closure, which is quite straightforward:

```
Definition closure:=
        (u,v:wrd) (R u v) -> (L u) -> (L v).
```

The definition of convergence is a bit more tricky, we use a variant of the accessibility (see above) property, it corresponds to Brouwer's *bar* concept ([8], [9]) of an element for a reduction. We call this property the "*unavoidability*". We shall say that `L` is unavoidable for a term `t` if all the reductions by `R` from `t` reach `L` in a finite number of steps.

In Coq such a definition is stated by the following definition:

```
Inductive unavoidable : wrd -> Prop :=
   (* If x belongs to L, then x is unavoidable *)
 | L_unavoidable:(x:wrd) (L x) -> (unavoidable x)
   (* If L is unavoidable for all the successors,
      then L is unavoidable *)
 | all_unavoidable:(x:wrd)
            ((y:wrd) (R x y) -> (unavoidable y))
          -> (unavoidable x).
```

We can now define the self-stabilization of `R`:

```
Definition self_stab:=
      closure /\ (w:wrd) (unavoidable w).
```

While this definition is more intuitive, we found it useful to base our technical development on alternative definitions which we present now. When the set of possible configurations is finite (which is generally the case since the set of possible state for each node is usually finite). `L` is unavoidable iff the reduction has no cycle with one node outside of `L`.

```
Definition no_cycle:= (w:wrd) (Rplus w w) -> (L w).
```

We can prove (using the `Auto` tactic) that `no_cycle` is equivalent to its negative version ("~" stands for the negation symbol).

```
Lemma no_cycle2:
       (w:wrd) no_cycle -> ~(L w) -> ~(Rplus w  w).
Auto.
Save.
```

With this second notion of convergence, we can re-formulate self-stabilization:

```
Definition self_stab2:=   closure /\ no_cycle.
```

It is possible to prove in Coq the equivalence of the two definitions.

Finally we give a third definition of convergence, that we will use in the following: `L` is unavoidable for `t` if the restriction of `R` to the complement $\overline{\mathtt{L}}$ of `L` terminates for `t`.

We define the restriction by the following line:

```
Definition restrict_R:=[x,y:wrd] ~(L y) /\ (R x y).
```

This leads to a third definition of self-stabilization:

```
Definition self_stab3:=
          closure /\ (w:wrd)(Acc restrict_R x).
End selfstab.
```

B. Definition of a distributed algorithm over a linear net

B.1 Definition of words

We do not show in detail the formalisation of words in Coq as it is independent of this work. The type of words is the type `List` of polymorphic lists that we load by the command `Require`.

```
Require Export PolyList.
(*
Inductive list [A : Set]  : Set :=
      nil : (list A)
    | cons : A->(list A)->(list A)
*)
Definition Word: Set -> Set := list.
```

`(app w w')` denotes the word obtained by concatenating `w` and `w'`, it is also noted `w ^ w'`.

B.2 Definition of sets of words

The notion of words set, which will be used in particular to define regular expressions, is also independent of this work, we just give the main operations we will need in the following:

A set of words is a predicate on words:

```
Definition wordset := Word -> Prop.
```

The set containing a single word is built by the `lword` operation (notice that `w = w1` is of type `Prop`):

```
Definition lword : Word->wordset :=
                           [w, w1 : Word] w = w1.
```

To read this last definition, remember that for a given word `w`, `(lword w)` has type `Word -> Prop`, i.e. `wordset`.

We give now the names of several operations allowing to built sets of words from other sets of words:

- By concatenation of words of two sets of words(concatenation of regular expression):
  `lconc:wordset -> wordset -> wordset`
- By concatenation of words of the same set(regular expression exponentiation):
  `lpow: nat -> wordset->wordset.`
- By arbitrary concatenation of words of the same set (* of regular expression):
  `lstar : wordset->wordset`
- By arbitrary (non null) concatenation of words of the same set($^+$ of regular expressions):
  `lplus : wordset->wordset`

B.3 Definition of a distributed algorithm

Now we give in Coq the definition of distributed algorithm given in section III-B, i.e. a rewriting system with three sets of rules: top(T), bottom(B) and middle(M) rules.

Formally, a rewriting system on words is a set of pairs $l_i \longrightarrow r_i$, where l_i and r_i are words. We shall say that a word w reduces to w' in one step if there exists a k such that $w = w_1 l_k w_2$ and $w' = w_1 r_k w_2$. l_k is called the *redex* of w in the rewriting step.

In our formalisation, we will define rewriting on words as the context closure of a relation corresponding to the rules $l_i \longrightarrow r_i$. As we said in section III-B, there are three kinds of rules, corresponding to three different context closure:

- For M the context closure is done on the left and on the right.
- For T it is done only on the left, because the rules of T apply only at the end of words,
- For B it is done only on the right, because the rules of B apply only at the beginning of words.

We formalise this in Coq by the following section:

```
Section reduct.
Variable A:Set.
```

We declare three relations corresponding to the three sets of rewriting rules.

```
Variables T,M,B: A -> A -> Prop.
```

Then we define the context closure corresponding to the rewriting system.

```
Inductive TBM_R: A -> A -> Prop :=
| T_T: (u,v,w:A) (T u v) -> (TBM_R (w^u) (w^v))
| B_B: (u,v,w:A) (B u v) -> (TBM_R (u^w) (v^w))
| M_M: (u,v,w,w':A)
       ~(w=nil) -> ~(w'=nil)
     -> (M u v) -> (TBM_R (w^(u^w')) (w^(v^w'))).
End reduct.
```

V. A METHOD FOR PROVING STABILIZATION: DEFINITIONS AND HYPOTHESIS

It is proved below (section VI) that under the hypothesis above, the convergence of a certain class of reductions, called the *top chains*, implies the convergence of the whole system. We precise now this class of reductions, we give alternatively definitions of concepts used in [1] and the corresponding definition that we made in Coq. All the following Coq definitions, as well as previous ones, have been done in a Coq library. This library will be used in the proofs of the first part, but also in the second.

A. Active words

In order to define the top chains, we need to define the notion of *active words*, which are basically words on a alphabet containing two versions of each letter: one active, one inactive. We shall signal active letters by underlying them. For example in the word $ab\underline{cd}ef$, the *positions* 3 and 4 are active, and the others are inactive. More formally:

Definition V-A.1 (Activated alphabet) Let E be an alphabet, we define the alphabet $\underline{E}$ containing the activated versions of the letters of E.

Definition V-A.2 (Activated word) Let E be an alphabet and $\underline{E}$ its activated version, *activated words* are the words on the alphabet $E \cup \underline{E}$.

Definition V-A.3 (Active word) A word containing an active letter is called active.

Definition V-A.4 (Inactive word) A word containing only inactive letters is called inactive.

B. Active words in Coq

In Coq we define active and inactive letters with an inductive type having two constructors applied to the type of (normal) letters.

```
Section act_inact.
Variable A:Set.

Inductive letter: Set :=
 |act  : A -> letter
 |inact: A -> letter.
```

At that point the type `A` of normal (not activated) letters is a parameter, we will instantiate this type by the type of natural number `nat` later when we need a concrete representation of letters. We also define the function `destr_letter` that returns the normal letter from an active or inactive letter. Notice that function definition syntax is very close to the syntax of functional languages.

```
Definition destr_letter :letter ->  A :=
 [l:letter]Cases l of
 |(act x) => x
 |(inact x) => x
 end.
```

We define now some notations:

- `|+x+|` for `(act x)`,
- `|-x-|` for `(inact x)`,
- `x::l` for `(cons x l)`,

Active words will be represented by the type `actword`:

```
Syntactic Definition actword :=(Word letter).
```

We give now some examples of definitions of useful functions on words, in particular `active_last` that allows to activate the last letter of a word, is defined by a recursive function (using the `Fixpoint` primitive of Coq).

```
Definition inactive_letter: letter -> letter:=
              [n:letter](inact (destr_letter n)).

Definition active_letter: letter -> letter :=
                [n:letter](act (destr_letter n)).


Definition deactivatel  :=  [l:letter]
 Cases l of
 | (inact e) => l
 | (act e) => (inact e)
 end.

Fixpoint deactivatew
 [w:(Word letter)]: (Word letter) :=
 Cases w of
   | nil => (nil letter)
   | (cons x u) =>
      (deactivatel x)::(deactivatew u)
 end.

Fixpoint active_last [x:actword]: actword:=
 Cases x of
 |  nil => nil
 |  (cons  (inact n) nil) => (act n)::nil
 |  (cons  (act n) nil) => (act n)::nil
 |  (cons x w) => x::(active_last w)
 end.
```

We define also some predicates on words, like `active_word_strict` that is such that `(active_word_strict w)` is provable if `w` contains an active letter, and `pure_active` that is such that `(pure_active w)` is provable if `w` contains no inactive letter.

```
Inductive active_word_strict: actword -> Prop:=
| aw_act: (u:actword)(c:A)
             (active_word_strict (|+c+| :: u))
| aw_ina: (u:actword) (c:letter)
        (active_word_strict u)
     -> (active_word_strict (c::u)).

Inductive pure_active:actword -> Prop :=
        | empty_pact:(pure_active nil)
        | aw_pact:(u:actword) (c:A) (pure_active u)
                        -> (pure_active (|+c|+ :: u)).
```

C. Active reductions

In order to define top chains, we need to define one more thing: *active reductions*, which is the extension to active words of a reductions over normal letters. Intuitively the following definitions explain how active positions are *propagated* during top chain reductions. We use the operation `deactivate` that returns the inactive version of a word.

Definition V-C.1: Let R be a rewriting system on words over the alphabet A, We extend R to activated words the following way:

Let u and u' be two inactive words such that $u \longrightarrow_R u'$ and P the set of positions of the redex of u. Let w be an active word such that $u = deactivate(w)$. then $w \longrightarrow_R w'$, where w' is defined by:

- $deactivate(w') = u'$ (i.e. w' and u' have the same letters);
- If $active\_pos(w) \cap P = \emptyset$, then $active\_pos(w') = active\_pos(w)$ (i.e. the redex remains inactive if it has no inactive letter).

else $active\_pos(w') = active\_pos(w) \cup P$. We shall say in this case that the active part of w has been *propagated* to all the redex.

For example, here are some examples of active reduction steps for the Ghosh's algorithm (the redex 30):

$$\begin{array}{ccc}
\#300\# & \longrightarrow & \#100\# \\
\#30\underline{0}\# & \longrightarrow & \#10\underline{0}\# \\
\#3\underline{0}0\# & \longrightarrow & \#\underline{10}0\# \\
\#3\underline{00}\# & \longrightarrow & \#\underline{100}\# \\
\#\underline{3}00\# & \longrightarrow & \#\underline{10}0\# \\
\#\underline{30}0\# & \longrightarrow & \#\underline{10}0\# \\
\#\underline{300}\# & \longrightarrow & \#\underline{100}\#
\end{array}$$

We can now define the top chain reductions, first we define the notion of *active step* of a reduction:

Definition V-C.2 (Active step) Let R be a rewriting system extended to active words. An active step of reduction of R is a rewriting step of R in which the rewritten redex is active.

Definition V-C.3 (Top chains) Let R be a rewriting system extended to active words. A top chain reduction of R is a reduction chain:

- starting by a word which only active letter is the top letter;
- such that every step is an active step.

For example, the following reduction of Ghosh's algorithm is a top chain:

$$\begin{array}{rl}
\#31\underline{2}\# & \longrightarrow_{M_2} \#3\underline{22}\# \\
 & \longrightarrow_{M_1} \#\underline{332}\# \\
 & \longrightarrow_{T1} \#\underline{330}\# \\
 & \longrightarrow_{M_2} \#\underline{300}\#
\end{array}$$

We see that the active part of the word grows until containing the whole word. *We shall see that if this kind of reduction converges to $\mathcal{L}$, then all reductions do.*

D. Active reductions in Coq

In Coq, We define the extension of a relation on words to activated words by defining the relation `actify` (on active words) *parameterized* by another relation (on inactive words), and such that `(actify R)` corresponds to `R` and propagates active regions according to the definition above.

This is done by the following inductive definition:

```
Inductive actify [A:Set; Ra: (A -> A -> Prop)]:
  A -> A -> Prop :=
| actify_ai: (u,v:A)
     (active_word u) -> (pure_active v)
  -> (Ra (deactivatew u) (deactivatew v))
  -> (actify Ra u v).
```

We also need to characterize relations that apply on *inactive* words only (see the declaration of T, B and M below). This done by the following predicate `Rel_inactive` on relations. It is such that if `(Rel_inactive R)` is provable, then `R` applies only on inactive words:

```
Definition Rel_inactive:=
[R:actword -> actword-> Prop]
 (u,v: actword) (R u v)
         -> (inact_word u) /\ (inact_word v).
End act_inact.
```

Now we can define active rewriting from three parameters: the three basic relation T, B and M, applying only on inactive words. The keyword `Hypothesis` is used to declare temporary hypothesis just like `Variable`, which means that for a particular instantiation of T, B, and M, they will have to be proved before we can use the definitions made in this section:

```
Section active_red.
Variable A:Set.
Syntactic Definition actword :=(Word (letter A)).
Variables T,B,M: actword -> actword -> Prop.

Hypothesis T_in: (Rel_inactive A T).
Hypothesis B_in: (Rel_inactive A B).
Hypothesis M_in: (Rel_inactive A M).
```

We define now the rewriting relation on inactive words, by applying `TBM_R` to `T`, `B` et `M`. We also define the restriction `R'` of `R` to elements outside of `L`.

```
(* Rewriting on inactive words *)
Definition R := (TBM_R A T B M).
(* Restriction of R to words outside L *)
Definition R' := (restrict_R A R L).
```

We also define restrictions of `R` to {`T`} (`Top`) and to {`M`,`B`} (`RminusTop`). This last restriction corresponds to the hypothesis 5 on the algorithm considered (see V-E).

```
Inductive Top: actword -> actword -> Prop :=
 | TopT: (u,v,w:actword) (T u v) -> (Top (w^u) (w^v)).

Inductive RminusTop:actword -> actword -> Prop :=
 | B_B': (u,v,w:actword)
             (B u v) -> (RminusTop (u^w) (v^w))
 | M_M': (u,v,w,w':actword)
       ~(w=nil) -> ~(w'=nil) -> (M u v)
            -> (RminusTop (w^(u^w')) (w^(v^w'))).
```

We define also the activated version of `R`, `Top` and `RminusTop` by applying `TBM_R` to the activated version of `T`, `B` and `M`:

```
(* Activated versions of T, B, M *)
Definition Tact:=(actify A T).
Definition Bact:=(actify A B).
Definition Mact:=(actify A M).

(* Active rewriting corresponding to the algorithm *)
Definition Ract := (TBM_R A Tact Bact Mact).

(* Restrictions of Ract: *)
Definition Ract' := (restrict_R A Ract L).

(* Active Top *)
Inductive Topact: actword -> actword -> Prop :=
 | TopTa: (u,v,w:actword)
              (Tact u v) -> (Topact (w^u) (w^v)).
```

Remark V-D.1: We can notice that if `R'`∪`Ract'` terminates, then `R` converges to `L`. This is because the union of the two relations simulates reductions over active and inactive words (a fortiori all possible reductions over normal words). This is stated by the lemma:

The main result used in TRC method is that it is sufficient to consider top chains only, i.e. it is sufficient that `Ract'` terminates on words which only active letter is the top letter.

E. Hypothesis

We state here the properties (from [1]) that need to be proved in order to use the TRC method on a particular algorithm R. We will see later how these properties are stated in Coq.

1. R is a rewriting system having the three sets of rules T, B, and M described in previous sections;
2. $\mathcal{L}$ is *closed* for R ;
3. R never stops outside of $\mathcal{L}$;
4. *R without the rules of T terminates.*

The last hypothesis is quite strong but seems to hold for a large class of self-stabilizing algorithms on linear networks (it is roughly that the algorithm verifies the *fairness* hypothesis for the top machine).

F. The hypothesis in Coq

We continue the previous Coq section and state the above hypothesis in Coq.

Hypothesis 1 is partly stated by the previous declarations, we need to state that R preserves length of words, and that it applies only on words of same size. To achieve this we first define :

- A property `same_length` stating that two words have the same length,
- A predicate `length_preserve` on relations stating that words related by a relation have to be of same length,
- A predicate `apply_same_length` stating that two relations apply on words of the same size (and only words of this size).

```
Inductive same_length: A -> A -> Prop:=
 | empty_empty: (same_length nil nil)
 | cons_length: (x,y:letter) (u,v:A)
                 (same_length u v)
             -> (same_length (cons x u) (cons y v)).

Definition length_preserve :=
   [R:A -> A -> Prop] (u,v:A) (R u v)
         -> (same_length u v).

Definition apply_same_length :=
   [R,Q:A -> A -> Prop] (u,v,w,x:A)
        (R u v) -> (Q w x) -> (same_length u w).
```

Then we use this predicates to state the hypothesis on T, B and M:

```
Hypothesis T_length_p:(length_preserve A T).
Hypothesis M_length_p:(length_preserve A M).
Hypothesis B_length_p:(length_preserve A B).

Hypothesis T_apply_length:(apply_same_length A T T).
Hypothesis M_apply_length:(apply_same_length A M M).
Hypothesis B_apply_length:(apply_same_length A B B).
Hypothesis apply_lengthTB:(apply_same_length2 A T B).
Hypothesis apply_lengthTM:(apply_same_length2 A T M).
Hypothesis apply_lengthBM:(apply_same_length2 A B M).
```

Hypothesis 2 is stated using the `closure` predicate defined in section IV-A.

```
Hypothesis R_closure:    (closure A R L).
Hypothesis Ract_closure: (closure A Ract L).
```

Hypothesis 3 (no deadlock) is stated by a simple existential property:

```
Hypothesis R_notL_red: (x:actword)
                    ~(L x) -> (EX y|(R x y)).
```

Finally hypothesis 4 is stated using the `Acc` predicate of section I-A.4.

```
Hypothesis RminusTop_finite:
       (x:actword) (finite actword RminusTop x).
```

VI. A METHOD FOR PROVING STABILIZATION: GENERIC PART

We prove in this section that in order to prove that a reduction system, for which hypothesis described above hold, converges to a set , it is sufficient to prove that top chain reductions converge to $\mathcal{L}$.

We will now explain shortly the classical proof of this result as it has been done in[1]. Then we will see how we manage to do it in Coq, despite it cannot be simply translated.

The proofs is in two steps:

1. A lemma (main lemma) is proved. It states a property on reduction systems.
2. Then the main lemma is use to prove that convergence of top chains is sufficient to prove converge of the whole system.

A. Main lemma

The lemma states that if two reduction systems terminate and semi-commute, then their union also terminates.

Lemma VI-A.1: Let R and Q be two reduction relations verifying the following semi-commutation property:

$$\forall x, y, z.(R\ x\ y) \wedge (Q\ y\ z) \Rightarrow \exists y'.(Q\ x\ y') \wedge (R^*\ y'\ z)$$

If R and Q terminate, then $R \cup Q$ also terminates.

Proof: The classical proof is by contradiction. Suppose that there exists an infinite reduction Δ by $R \cup Q$, starting with a term t. There we can distinguish two cases (notice we use here the excluded middle law on a non decidable property):

- Either there is a finite number of Q steps, then there exists an element u of Δ after which there are only R steps. This is a contradiction since it means that there exists an infinite reduction by R from u.
- Or there is an infinite number of Q steps, then we can build an infinite reduction Δ' by "shifting" Q steps to the beginning of the reduction. We obtain an infinite reduction by Q, which is also a contradiction

Thus there cannot be an infinite reduction by $R \cup Q$. ■

This proof, using the excluded middle law on the undecidable property "to have an infinite number of Q steps", is therefore not constructive. Non constructive reasoning cannot be done in Coq without giving additional axioms. It turns out that it is possible to prove constructively this property. This is the subject of the following section.

B. Main lemma in Coq

As usual we start by opening a section and by declaring temporary objects and hypothesis. first we declare two relations `R` and `Q` on a type `A`.

```
Section union_sec.
Variable A : Set.
Variables R, Q :  A -> A ->  Prop.
Definition RUQ := (union R Q).
```

First hypothesis: `Q` and `R` are terminating

```
Hypothesis R_Acc : (x : A)  (Acc A R x).
Hypothesis Q_Acc : (x : A)  (Acc A Q x).
```

`R` and `Q` commute:

```
Hypothesis R_Q_commute: (x,y,z:A)
         (R x y) -> (Q y z)
    ->  (Ex [y' : A] (Q x y') /\ (R y' z)).
```

Then we prove that the union RUQ of R and Q is terminating. We use well founded induction on the reductions on R and Q.

```
Theorem RUQ__commut_Acc: (x : A)  (Acc A RUQ x).
Intros x.
Elim x  using (well_founded_ind A R R_Acc).
Clear x.
Intro x.
Elim x  using (well_founded_ind A Q Q_Acc).
Intros x0 H H0.
Apply Acc_intro.
Intros y H1.
Inversion_clear H1; Auto with *.
Apply H; Auto with *.
Intros y0 H1.
Elim (R_Q_commute y0 y x0 H1 H2); Auto with *.
Intros x1 H3; Elim H3; Intros H4 H5; Clear H3.

Generalize (H0 x1 H5).
Intros H3.
Inversion_clear H3; Auto with *.
Apply H6.
Unfold RUQ explicit_union.
Right.
Assumption.
Save.

End union_sec.
```

C. *Restriction to top chains*

We are now able to prove that the convergence of top chains is sufficient to prove the convergence of an algorithm. We will directly prove this result in Coq.

Since R′ and Ract′ are the restrictions of R and Ract to words that do not belong to L, it is easy to prove that it is sufficient that R′ URact′ terminate to prove that R converges to L. We do not show the proof script, only the statement of the property (that we declare as a theorem).

```
Theorem restrict_notinL: (x:actword)
 (Acc  (union R' Ract') x)
   -> (unavoidable R' x).
...
Save.
```

Therefore we have now to prove that it is sufficient that the top chains restricted to elements outside L terminate to prove that R′ URact′ terminates. First we prove that R′ URact′ terminates on words which only active letter is the last letter, then R′ URact′ terminates on all words.

```
Lemma lastact_only: (x:actword)
 ((Acc  (union R' Ract') (active_last ? x)))
   -> (Acc  (union R' Ract') x).
...
Save.
```

The proof of this lemma necessitates a long development in Coq in order to use some particular properties on words. We do not show the proof here, see [1] for details. We can now use this result and the main lemma to conclude:

```
Theorem topc:
 ((x:actword)
  (last_active x) -> (Acc actword Ract'  x)
 )
  -> (x:actword) ~x=nil
  -> (Acc actword (union R' Ract') x).
Intros hyp x hyp2.
Apply lastact_only.
Apply RUQ__commut_Acc. (* main lemma *)
...
Save.
```

VII. A METHOD FOR PROVING STABILIZATION: SPECIFIC PART

We give now a short description of the use of the generic Coq development described above. The idea is the following: to prove self-stabilization of a particular algorithm A, if we can prove that top chains (restricted to elements outside $\mathcal{L}$) terminate, then by applying the theorems topc and restrict_notinL proved in previous section, *we obtain a complete and type-checkable (thus certified) proof of the convergence of* R.

A. *An example of algorithm*

We take for the example a very simple algorithm, there are only two possible states for each node: 0 or 1.

The three sets of rules T, B and M are the following:

- T:
$$00 \longrightarrow 01$$
$$10 \longrightarrow 11$$
- B:
$$00 \longrightarrow 01$$
$$11 \longrightarrow 01$$
$$10 \longrightarrow 11$$
- M:
$$00 \longrightarrow 01$$
$$10 \longrightarrow 11$$

The set $\mathcal{L}$of legitimate configurations is:

$$\mathcal{L} = \{01^+\}$$

```
Require Omega. (* library of integers *)
Syntactic Definition actword :=(Word (letter nat)).
```

We define two sets of words corresponding to the two simple regular expressions $\underline{1}$ and $\underline{0}$ using the operation lword defined in section IV-B.2.

```
Definition set_one:=
       (lword (letter nat) (|+(1)+| :: nil)).
Definition set_zero:=
       (lword (letter nat) (|+(0)+| :: nil)).
```

Then we define inductive predicates (i.e. sets) on words corresponding to more complex regular expressions: $\underline{1}^*, \underline{1}^+, \underline{0}\,\underline{1}^+, \underline{0}\,\underline{1}^+$ that will be used in the following, we show here some of these definitions.

```
Definition Onestar := (lstar (letter nat) set_one).
Definition Oneplus := (lplus (letter nat) set_one).
Inductive zero_one_plus:actword -> Prop:= (* 01+ *)
   zunpl:(u:actword) (Oneplus u)
              -> (zero_one_plus (|+(0)+| :: u)).
...
```

We define $\mathcal{L}$ by the regular expression 01^+:

```
Definition ex1L:=zero_one_plus.
```

We declare the three rewriting systems T, B and M as the three following inductive relations:

```
Inductive ex1T: actword -> actword -> Prop :=
  | ex1T1:(ex1T (|-(0)-| :: (|-(0)-| :: nil))
                (|-(0)-| :: (|-(1)-| :: nil)))
  | ex1T3:(ex1T (|-(1)-| :: (|-(0)-| :: nil))
                (|-(1)-| :: (|-(1)-| :: nil))).

Inductive ex1B: actword -> actword -> Prop :=
  | ex1B1:(ex1B (|-(0)-| :: (|-(0)-| :: nil))
                (|-(0)-| :: (|-(1)-| :: nil)))
  | ex1B2:(ex1B (|-(1)-| :: (|-(1)-| :: nil))
                (|-(0)-| :: (|-(1)-| :: nil)))
  | ex1B3:(ex1B (|-(1)-| :: (|-(0)-| :: nil))
                (|-(1)-| :: (|-(1)-| :: nil))).

Inductive ex1M: actword -> actword -> Prop :=
  | ex1M1:(ex1M (|-(0)-| :: (|-(0)-| :: nil))
                (|-(0)-| :: (|-(1)-| :: nil)))
  | ex1M2:(ex1M (|-(1)-| :: (|-(0)-| :: nil))
                (|-(1)-| :: (|-(1)-| :: nil))).
```

We instantiate the definitions of `R`, `Ract`, `R'`, `Ract'` with the specific definitions of the algorithm.

```
Syntactic Definition ex1R :=(R ex1T ex1M ex1B).
Syntactic Definition ex1Ract :=(Ract ex1T ex1M ex1B).
Syntactic Definition ex1R' :=(R' ex1T ex1M ex1B ex1L).
Syntactic Definition ex1Ract' :=(Ract' ex1T ex1M ex1B ex1L).
```

To be able to apply the result of the generic part (`restrict_notinL` and `topc`), we need to prove that T, B and M verify all the hypothesis declared previously. As an example we show the proof that `ex1T` applies on inactive words:

```
Lemma T_relinact:(Rel_inactive ? ex1T).
Red.
Intros.
Elim H;Split;Auto.
Save.
```

We also have to prove the closure of `ex1L` by the reduction `ex1R` (we do not show the proof).

```
Lemma ex1L_stable:(u:actword)
 (ex1L u) -> (v:actword) (ex1Ract u v) -> (ex1L v).
...
Save.
```

Once all hypothesis have been verified and that the generic theorems have been applied, it remains to prove the property that top chain reductions converge. This is done by using a rewriting technique: *narrowing*. We cannot describe this method here. Again, see[1] for a detailed description of a general method to prove this property.

VIII. Conclusion

We have use effectively a general purpose proof assistant to implement a generic proof of self-stabilization. The generic part of the method has been proved in Coq and can be reused and instantiated on particular algorithms. Parts of the development may be used to implement other methods, in particular the technical development (not shown in this work) on words.

The medium term goal of this work is to propose a proof environment dedicated to self-stabilization. In this framework, the specific work that has to be done for each specific algorithm is still to be (semi-)automated. The main idea is to provide a tool that generates from the definition of a particular algorithm:

- Proof obligations, expressed in the Coq formalism, implying the convergence of the top chains reductions.
- A *proof* (i.e. a CIC term) that if this proof obligations are satisfied, then the top chains are converging to $\mathcal{L}$.

It seems that the first step can't be fully automated and will need user interaction. In particular in [1], the technique used to prove the specific part of the proof requires the user to guess some regular expressions.

IX. Thanks

I am extremely grateful to Laurence Puel, Joffroy Beauquier and Frederick Magniette for the numerous discussions we had, the explanations they gave to me and more generally for their precious help. I thank also Gilles Barthes for helping me writing this paper and Loïc Pottier for helping me simplifying some proofs of the Coq development.

References

[1] L. Fribourg J. Beauquier, B. Bérard and F. Magniette, "Proving convergence of self-stabilizing systems using first-order rewriting and regular languages," *Distributed Computing*, vol. 14, no. 2, pp. 83–95, 2001.
[2] "The Coq Proof Assistant," `http://coq.inria.fr/`.
[3] William A. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Roger Hindley Jonathan P. Seldin, Ed., pp. 479–490. Academic Press, London, 1980.
[4] Morten Heine. Sørensen and P. Urzyczyn, "Lectures on the curry-howard isomorphism," Available as DIKU Rapport 98/14, 1998.
[5] Thierry Coquand and Christine Paulin-Mohring, "Inductively defined types," in *Proceedings of Colog'88*, P. Martin-Löf and G. Mints, Eds. 1990, vol. 417 of *Lecture Notes in Computer Science*, Springer-Verlag.
[6] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.
[7] S. Ghosh, "An alternative solution to a problem of system stabilization," in *Proceedings of TOPLAS*. 1993, pp. 735–742, ACM Press.
[8] E. Martino and P. Giaretta, "Dummett and the bar theorem," 1979.
[9] A. Heyting, *Intuitionism: An Introduction*, North-Holland, Amsterdam, 1956.