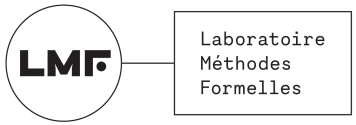
Vérification déductive de programmes avec Why3

Jean-Christophe Filliâtre
CNRS

Université Saint-Joseph de Beyrouth

17 juin 2021



Software is hard. – Don Knuth

pourquoi ?

- mauvaise interprétation des spécifications
- programmation dans l'urgence
- changements incompatibles
- logiciel = objet très complexe
- etc.

un exemple célèbre : *binary search*

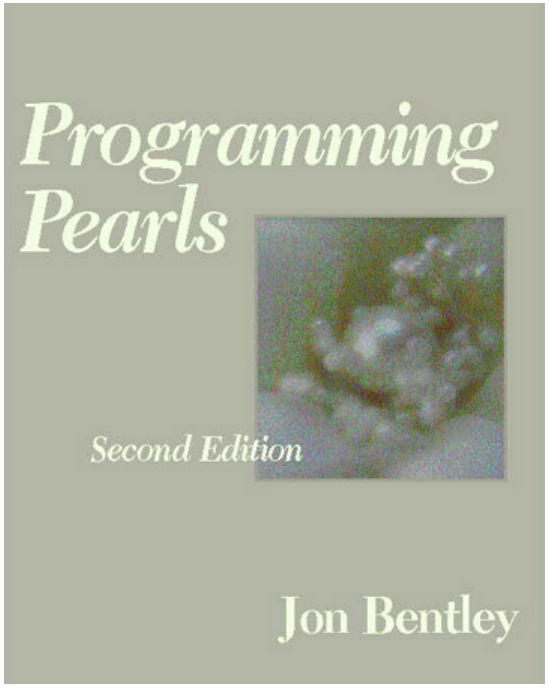
étant donné un tableau trié

-7	-1	2	2	42	987	1729

déterminer si une certaine valeur y apparaît

un exemple célèbre : *binary search*

première publication en 1946
première publication sans bug en 1962



Jon Bentley. Programming Pearls. 1986.

Writing correct programs

the challenge of binary search

et pourtant...

en 2006, un bug a été trouvé dans le code de *binary search* de la bibliothèque standard de Java

> Joshua Bloch, Google Research Blog
> *"Nearly All Binary Searches and Mergesorts are Broken"*

ce bug était là depuis 9 ans

```
...
int mid = (low + high) / 2;
int midVal = a[mid];
...
```

peut provoquer un débordement de capacité arithmétique,
suivi d'un accès en dehors des bornes du tableau

un correctif possible

```
int mid = low + (high - low) / 2;
```

de meilleurs langages de programmation

- meilleure syntaxe
  (éviter de considérer `DO 17 I = 1. 10` comme une affectation)

- plus de typage
  (éviter de confondre des mètres et des yards)

- plus d'avertissements du compilateur
  (éviter d'oublier certains cas)

- etc.

le test systématique et rigoureux est une autre réponse, complémentaire

mais le test est

- coûteux
- parfois très difficile à mettre en œuvre
- et surtout incomplet (à de très rares exceptions près)

les méthodes formelles proposent une approche mathématique de la correction du logiciel

qu'est-ce qu'un programme ?

il y a plusieurs aspects en jeux

- ce que l'on calcule (quoi)
- la manière de le calculer (comment)
- la raison pour laquelle c'est correct (pourquoi)

qu'est-ce qu'un programme ?

le programme, ce n'est que le « comment », et rien d'autre

le « quoi » et le « pourquoi » n'en font pas partie

ce sont des cahiers des charges, des commentaires, des pages web, des croquis, des articles de recherche, etc.

- comment : 2 lignes de C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- comment : 2 lignes de C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- quoi : 15 000 décimales de π

- pourquoi : beaucoup de maths, dont

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2\, 2^{i+1}}{(2i+1)!}$$

les méthodes formelles proposent une approche rigoureuse de la programmation, où on se donne

- une spécification écrite dans un langage mathématique
- une preuve que le programme vérifie cette spécification

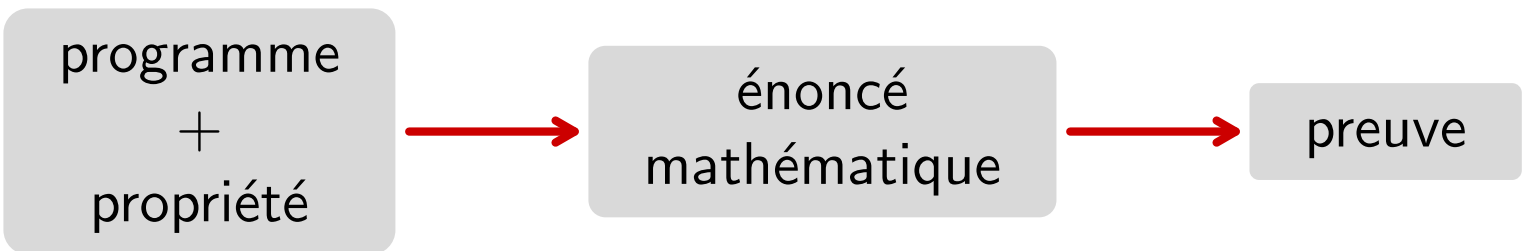
que souhaite-t-on prouver ?

- sûreté : le programme ne ≪ plante ≫ pas
  - pas d'accès illégal à la mémoire
  - pas d'opération illégale, comme une division par zéro
  - le programme termine

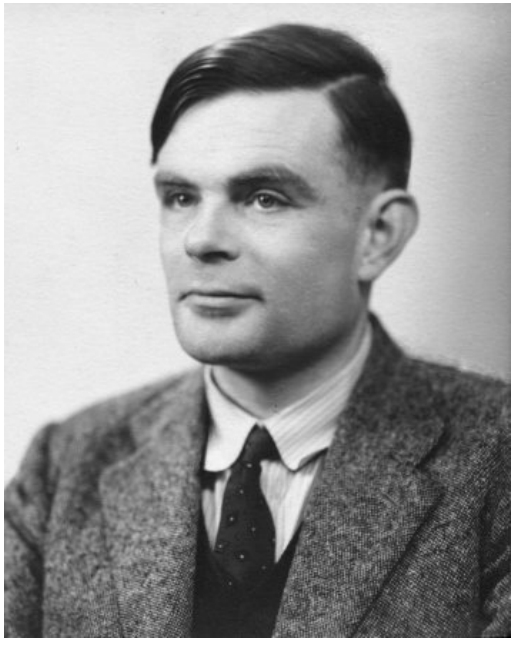
- correction fonctionnelle
  - le programme fait ce qu'il est censé faire

on peut citer le model checking, l'interprétation abstraite, etc.

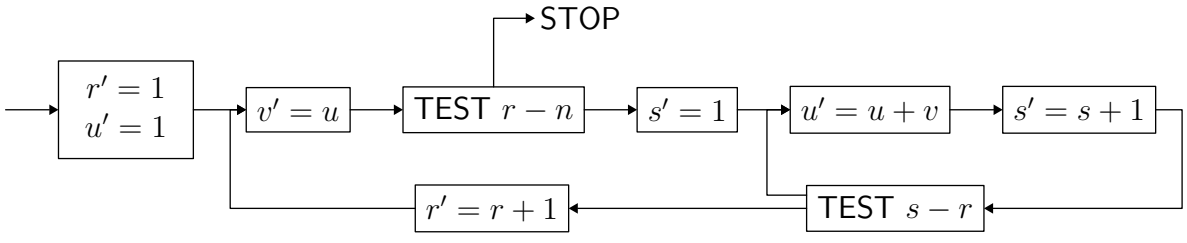
cet exposé présente la vérification déductive



ce n'est pas nouveau

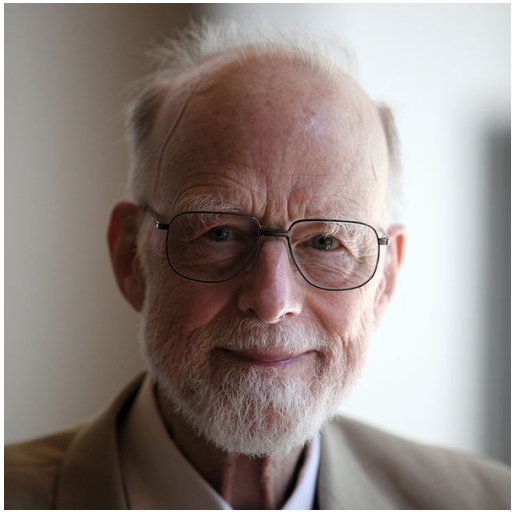


A. M. Turing. Checking a large routine. 1949.



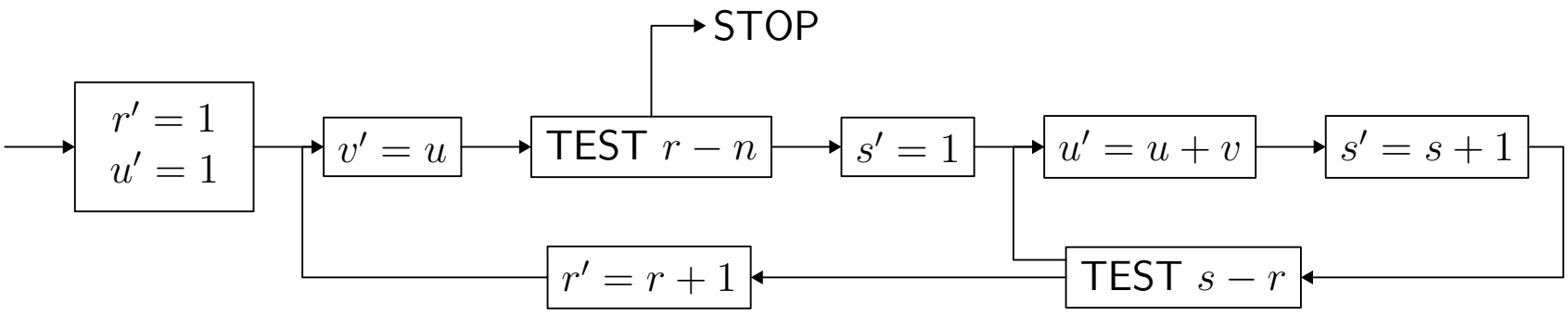
ce n'est pas nouveau

Robert Floyd.
Assigning Meanings to Programs.
1967

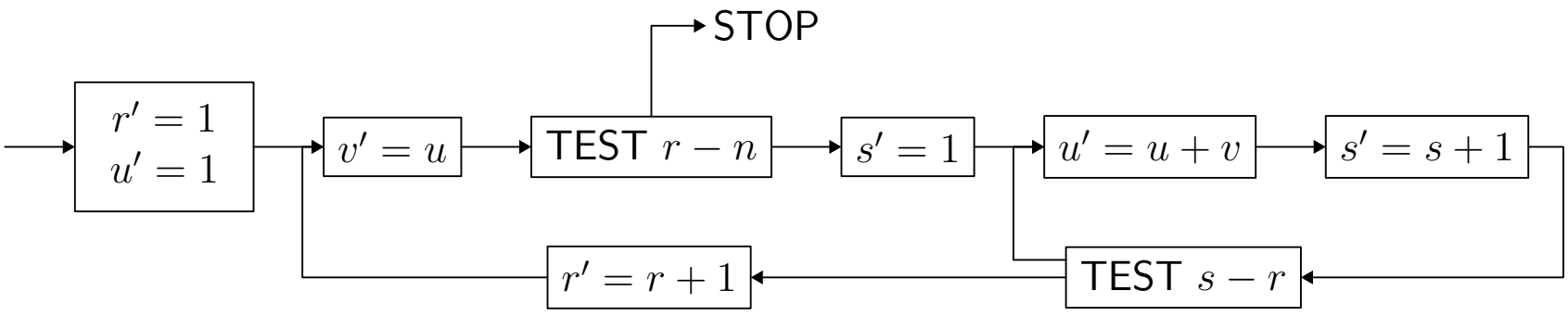


Tony Hoare.
An Axiomatic Basis for Computer Programming.
1969

checking a large routine (Turing, 1949)

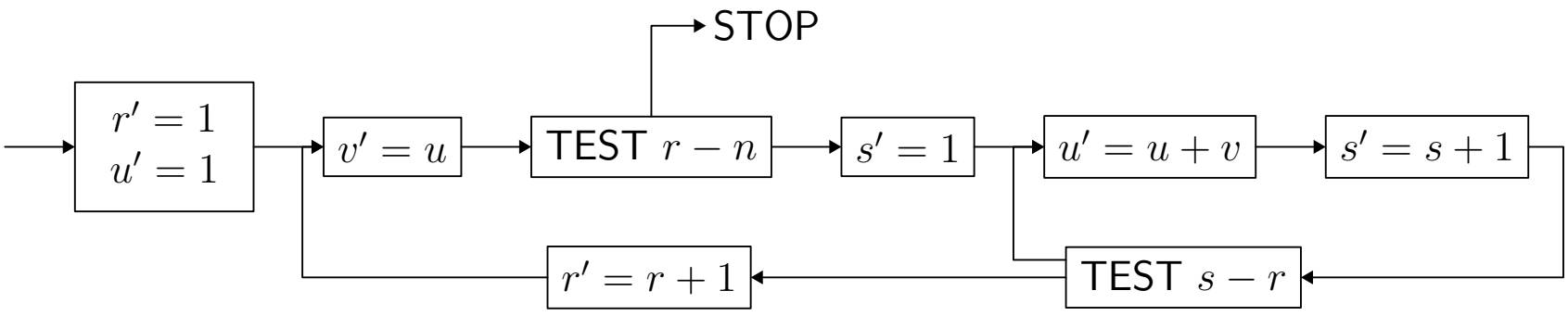


checking a large routine (Turing, 1949)



$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$

checking a large routine (Turing, 1949)



précondition $\{n \geq 0\}$

$u \leftarrow 1$

`for` $r = 0$ `to` $n - 1$ `do`

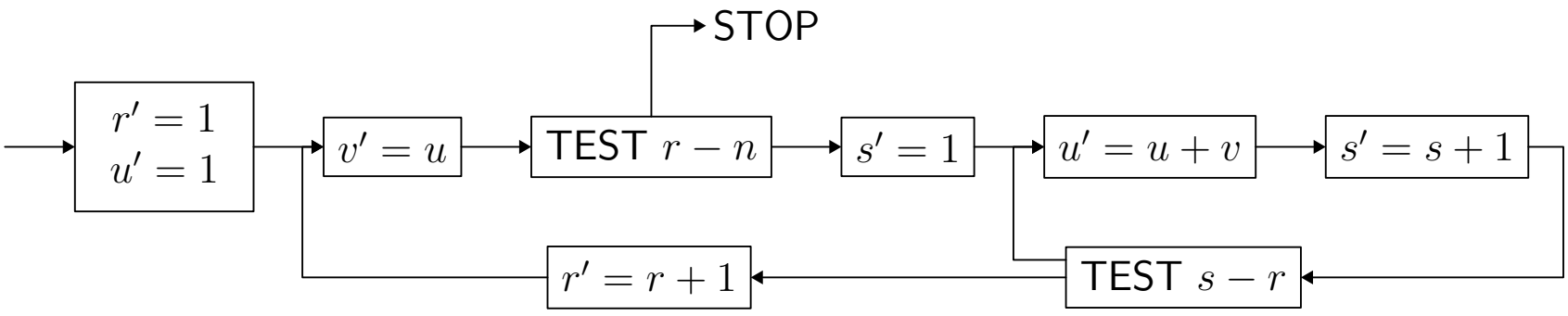
  $v \leftarrow u$

  `for` $s = 1$ `to` r `do`

    $u \leftarrow u + v$

postcondition $\{u = n!\}$

checking a large routine (Turing, 1949)



précondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do` invariant $\{u = r!\}$
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do` invariant $\{u = s \times r!\}$
    $u \leftarrow u + v$
postcondition $\{u = n!\}$

condition de vérification

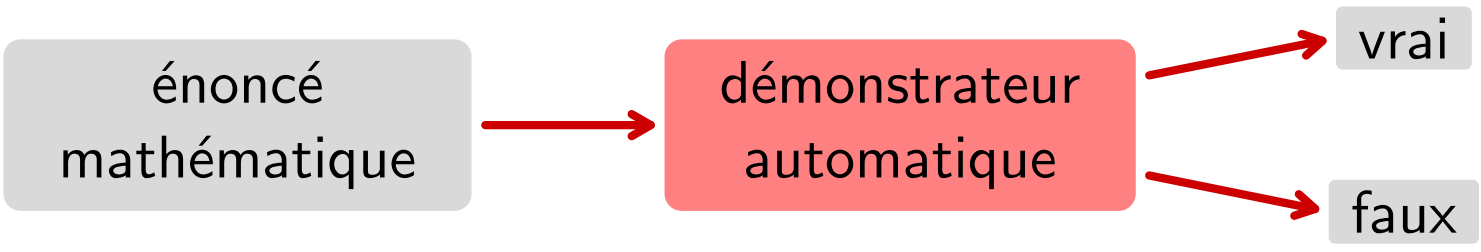
```
forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = n!) /\
  (0 <= n - 1 ->
    1 = 0! /\
    (forall u:int.
      (forall r:int. 0 <= r /\ r <= n - 1 -> u = r! ->
        (1 > r -> u = (r + 1)!) /\
        (1 <= r ->
          u = 1 * r! /\
          (forall u1:int.
            (forall s:int. 1 <= s /\ s <= r -> u1 = s * r! ->
              (forall u2:int.
                 u2 = u1 + u -> u2 = (s + 1) * r!)) /\
            (u1 = (r + 1) * r! -> u1 = (r + 1)!)))) /\
      (u = ((n - 1) + 1)! -> u = n!)))
```

que faire de cet énoncé mathématique ?

bien sûr, on pourrait le prouver à la main (comme Turing et Hoare)
mais c'est long, fastidieux, sujet à de nombreuses erreurs

aussi, on se tourne vers des outils qui mécanisent le raisonnement mathématique

démonstration automatique

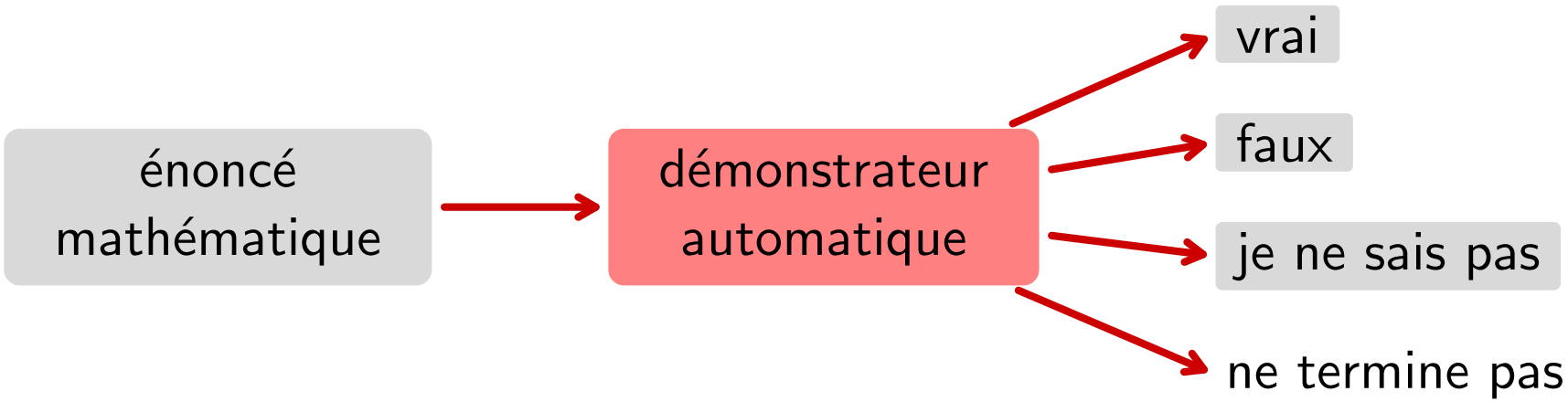


il n'est pas possible d'écrire un tel programme
(Turing/Church, 1936, d'après Gödel)

c'est le théorème anti-chômage pour les mathématiciens

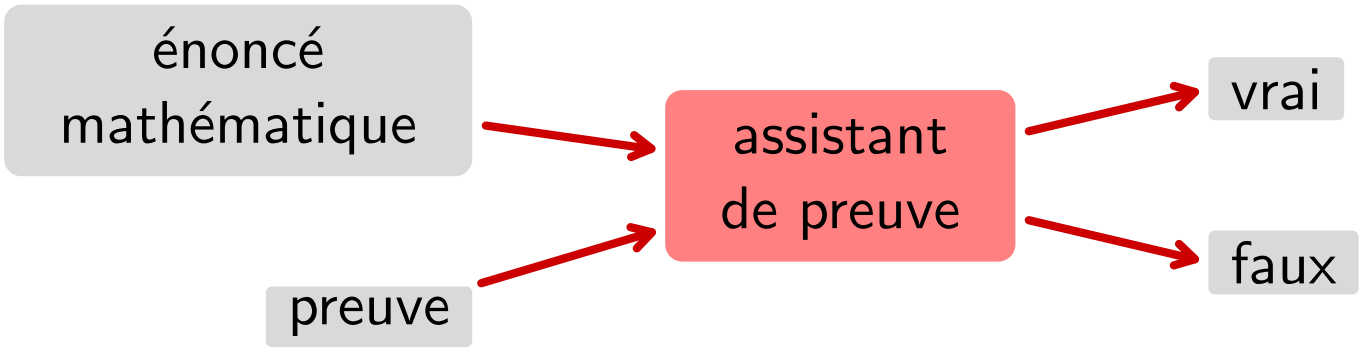
Kurt Gödel

démonstration automatique



exemples : Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

si on se contente de vérifier une preuve, cela redevient décidable



exemples : Coq, Isabelle, PVS, HOL-light, etc.

Why3, un outil de vérification déductive

idée centrale : utiliser le plus grand nombre possible de démonstrateurs, tant automatiques qu'interactifs

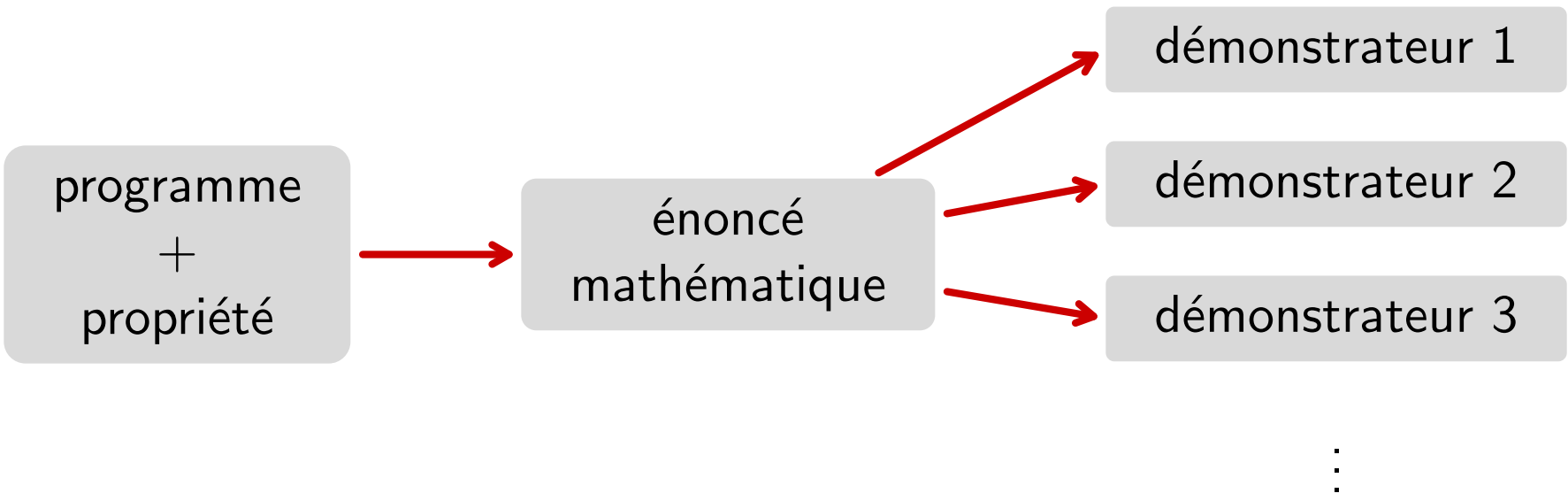
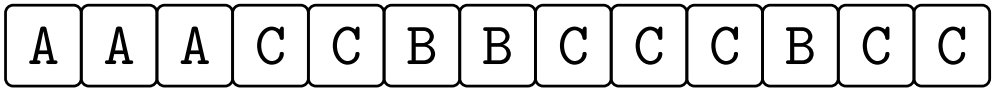


illustration sur un exemple

un ensemble de N votes est donné dans un tableau



objectif : déterminer si un candidat obtient la majorité absolue

une solution élégante

due à Boyer & Moore (1980)

en temps linéaire

en espace constant

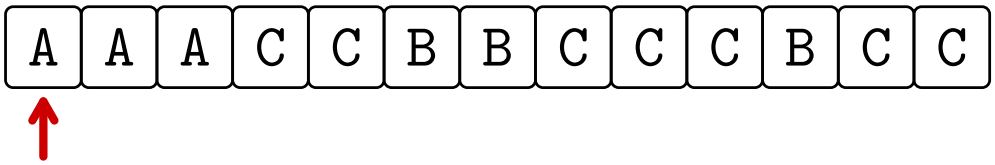
MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

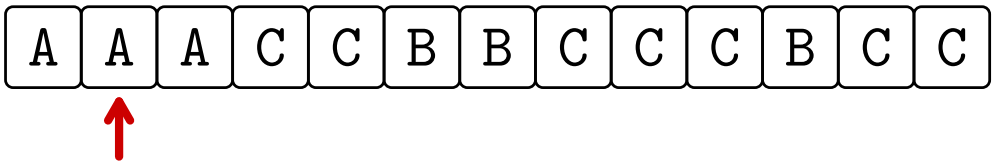
Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

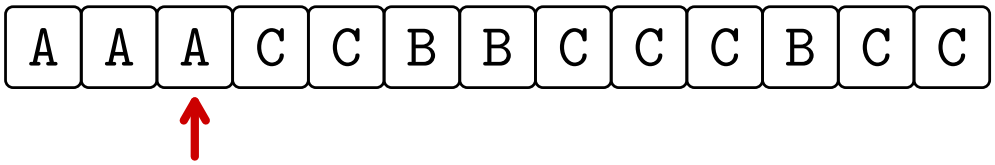
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.



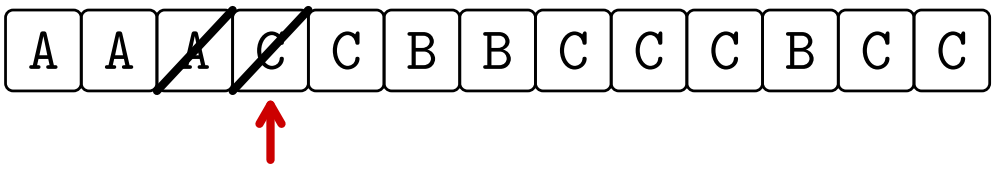

```
cand = A
k    = 1
```



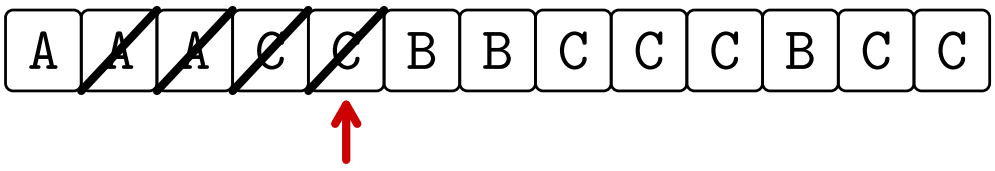

```
cand = A
k    = 2
```



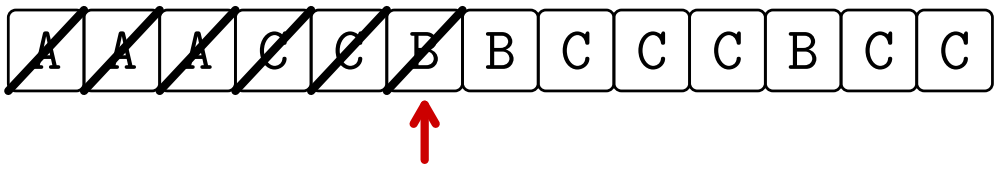

```
cand = A
k    = 3
```



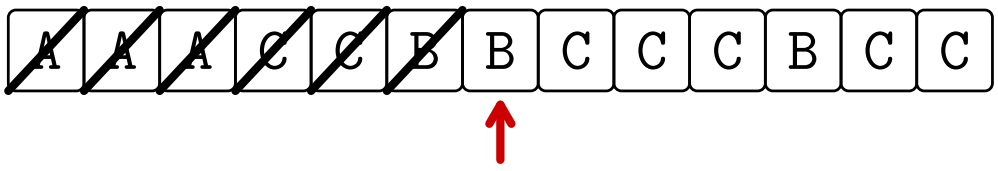

```
cand = A
k    = 2
```



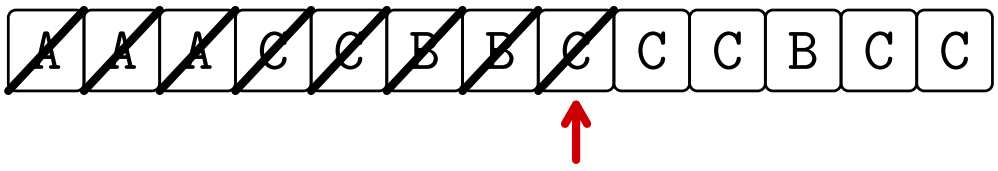

```
cand = A
k    = 1
```



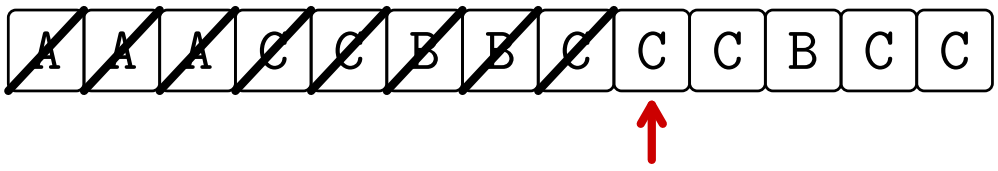

```
cand = A
k    = 0
```



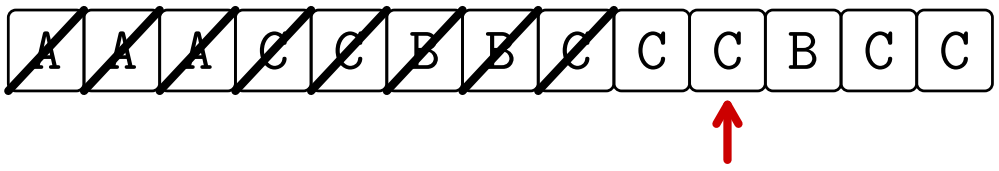

```
cand = B
k    = 1
```



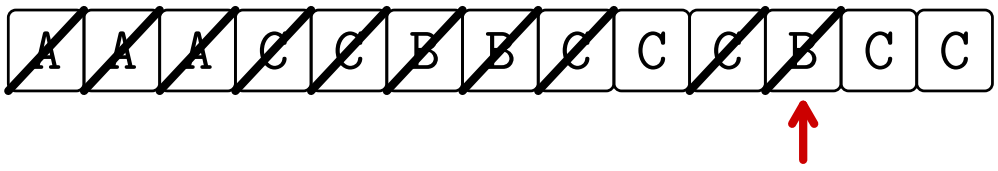

```
cand = B
k    = 0
```



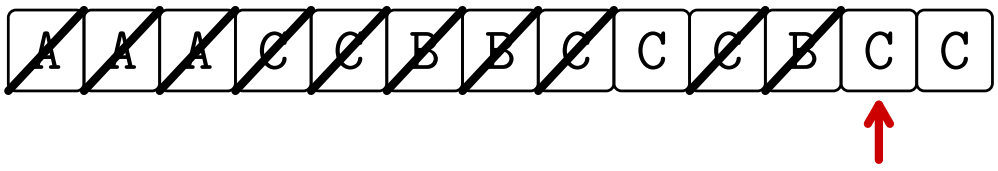

```
cand = C
k    = 1
```



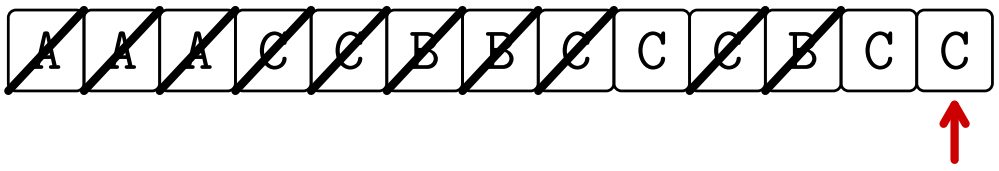

```
cand = C
k    = 2
```



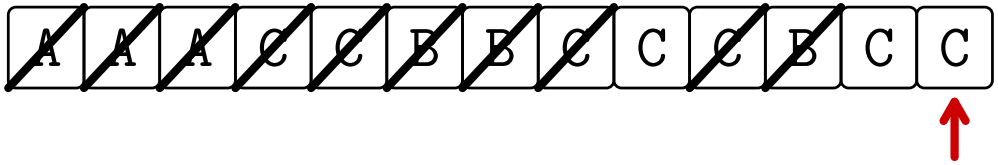

```
cand = C
k    = 1
```




```
cand = C
k    = 2
```




```
cand = C
k    = 3
```




```
cand = C
k    = 3
```

puis on vérifie si C a effectivement la majorité,
avec une seconde passe
dans ce cas, c'est vrai : $7 > 13/2$

code en Fortran

```
      SUBROUTINE MJRTY(A, N, BOOLE, CAND)
      INTEGER N
      INTEGER A
      LOGICAL BOOLE
      INTEGER CAND
      INTEGER I
      INTEGER K
      DIMENSION A(N)
      K = 0
C     THE FOLLOWING DO IMPLEMENTS THE PAIRING PHASE. CAND IS
C     THE CURRENTLY LEADING CANDIDATE AND K IS THE NUMBER OF
C     UNPAIRED VOTES FOR CAND.
      DO 100 I = 1, N
      IF ((K .EQ. 0)) GOTO 50
      IF ((CAND .EQ. A(I))) GOTO 75
      K = (K - 1)
      GOTO 100
50    CAND = A(I)
      K = 1
      GOTO 100
75    K = (K + 1)
100   CONTINUE
      IF ((K .EQ. 0)) GOTO 300
      BOOLE = .TRUE.
      IF ((K .GT. (N / 2))) RETURN
C     WE NOW ENTER THE COUNTING PHASE. BOOLE IS SET TO TRUE
C     IN ANTICIPATION OF FINDING CAND IN THE MAJORITY. K IS
C     USED AS THE RUNNING TALLY FOR CAND. WE EXIT AS SOON
C     AS K EXCEEDS N/2.
      K = 0
      DO 200 I = 1, N
      IF ((CAND .NE. A(I))) GOTO 200
      K = (K + 1)
      IF ((K .GT. (N / 2))) RETURN
200   CONTINUE
300   BOOLE = .FALSE.
      RETURN
      END
```

code en Why3

```
let mjrty (a: array candidate) : candidate
= let n = length a in
  let ref cand = a[0] in
  let ref k = 0 in
  for i = 0 to n - 1 do
    if k = 0 then begin cand <- a[i]; k <- 1 end
    else if cand = a[i] then k <- k + 1 else k <- k - 1
  done;
  if k = 0 then raise Not_found;
  if 2 * k > n then return cand;
  k <- 0;
  for i = 0 to n - 1 do
    if a[i] = cand then begin
      k <- k + 1;
      if 2 * k > n then return cand
    end
  done;
  raise Not_found
```

prouvons ce programme

- précondition

```
let mjrty (a: array candidate)
  requires { 1 <= length a }
```

- postcondition en cas de succès

```
  ensures
    { 2 * numof a result 0 (length a) > length a }
```

- postcondition en cas d'échec

```
  raises { Not_found ->
           forall c: candidate.
           2 * numof a c 0 (length a) <= length a }
```

première boucle

```
for i = 0 to n-1 do
  invariant { 0 <= k <= numof a cand 0 i }
  invariant { 2 * (numof a cand 0 i - k) <= i - k }
  invariant { forall c: candidate. c <> cand ->
                2 * numof a c 0 i <= i - k  }
  ...
```

seconde boucle

```
for i = 0 to n-1 do
  invariant { k = numof a cand 0 i }
  invariant { 2 * k <= n }
  ...
```

la condition de vérification exprime

- la sûreté
  - accès dans les bornes du tableau
  - terminaison
- respect des spécifications
  - les invariants sont initialisés et préservés
  - les postconditions sont établies

elle est entièrement prouvée par un démonstrateur automatique

terminaison

un programme peut ne pas terminer

dans le langage de Why3, on a

- des boucles `while`

```
while lo <= hi do ...
```

- des fonctions récursives

```
let rec f (n: int) : int = ...
```

on peut prouver

la correction partielle
si la précondition est vraie
et si le programme termine
alors la postcondition est vérifiée

ou

la correction totale
si la précondition est vraie
alors le programme termine
et la postcondition est vérifiée

la correction partielle est une propriété très faible

l'absence de terminaison peut rendre une preuve totalement vaine

comment prouver la terminaison

mauvaise nouvelle : on ne peut pas vérifier automatiquement si un programme termine

il faut fournir un argument, comme par exemple une borne supérieure sur le nombre d'itérations / d'appels récursifs

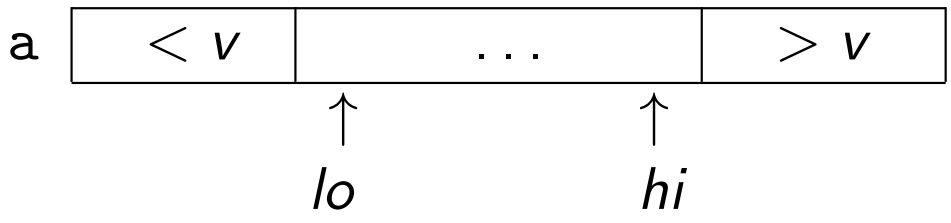
en Why3, on l'indique avec un variant

```
while lo <= hi do
  variant { hi - lo }
  ...
```

```
let rec f (n: int) : int
  variant { 101 - n }
= ...
```

I do not think it means what you think it means

```
lo ← 0
hi ← length a − 1
while lo ≤ hi do
   m ← lo + (hi − lo)/2
   if a[m] < v
      lo ← m + 1
   else if a[m] > v
      hi ← m − 1
   else
      return m
return -1
```



```
let binary_search a v
  requires ... le tableau est trié ...
  ensures 0 ≤ result < length a ∧ a[result] = v
          ∨ result = −1 ∧ ∀i. 0 ≤ i < length a ⇒ a[i] ≠ v
```

c'est tout à fait correct

mais si on écrit

```
let binary_search a v
  requires ... le tableau est trié ...
  ensures (0 ≤ result < length a ⇒ a[result] = v)
          ∧ (result = −1 ⇒ ∀i. 0 ≤ i < length a ⇒ a[i] ≠ v)
```

alors le programme peut maintenant renvoyer −2 et être vérifié !

et si on écrit

```
let binary_search a v
  requires ... le tableau est trié ...
  ensures 0 ≤ result < length a ⇒ a[result] = v
          ∧ result = −1 ⇒ ∀i. 0 ≤ i < length a ⇒ a[i] ≠ v
```

(noter que les parenthèses ont été supprimées)

alors le programme peut maintenant renvoyer 42 et être vérifié !

avant de faire une preuve, il faut avoir la bonne spécification

puis s'assurer qu'elle est acceptée par notre interlocuteur

sinon, la preuve est une perte de temps

code fantôme

des données et du code ajoutés à notre programme
pour rendre la preuve plus simple

on cherche le plus petit nombre de Fibonacci supérieur ou égal à n

$a, b \leftarrow 0, 1$
`while` $a < n$ `do`
  $a, b \leftarrow b, a + b$
`return` a

pour le vérifier,
on peut introduire un invariant de boucle comme celui-ci

$a, b \leftarrow 0, 1$
`while` $a < n$ `do`
  `invariant` $\exists i.\ i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
`return` a

mais prouver l'existence de i est difficile pour les démonstrateurs

on garde trace de i dans une variable fantôme

$a, b \leftarrow 0, 1$
$i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  $i \leftarrow i + 1$
`return` a

plutôt que de laisser le démonstrateur chercher la valeur de i,
on la fournit

les règles du jeu

- le code fantôme peut lire les données du programme mais pas les modifier
- le code fantôme ne peut pas modifier le flot de contrôle du programme
- le programme ne voit pas les données fantômes

conséquence : le code fantôme peut être supprimé sans modification observable

$a, b \leftarrow 0, 1$
$i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  $i \leftarrow i + 1$
`return` a

$a, b \leftarrow 0, 1$
`ghost` $i \leftarrow 0$
`while` $a < n$ `do`
  `invariant` $i \geq 0 \wedge a = F_i \wedge b = F_{i+1}$
  $a, b \leftarrow b, a + b$
  `ghost` $i \leftarrow i + 1$
`return` a

$a, b \leftarrow 0, 1$

`while` $a < n$ `do`

$\quad a, b \leftarrow b, a + b$

`return` a

supposons que l'on cherche à montrer que, pour tout n,

$$n! \geq 1$$

on peut écrire un programme qui en fait la preuve

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`

$\quad f \leftarrow i \times f$

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)!$
  $f \leftarrow i \times f$
`assert` $f = n!$

$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)! \wedge f \geq 1$
  $f \leftarrow i \times f$
`assert` $f = n!$
`assert` $n! \geq 1$

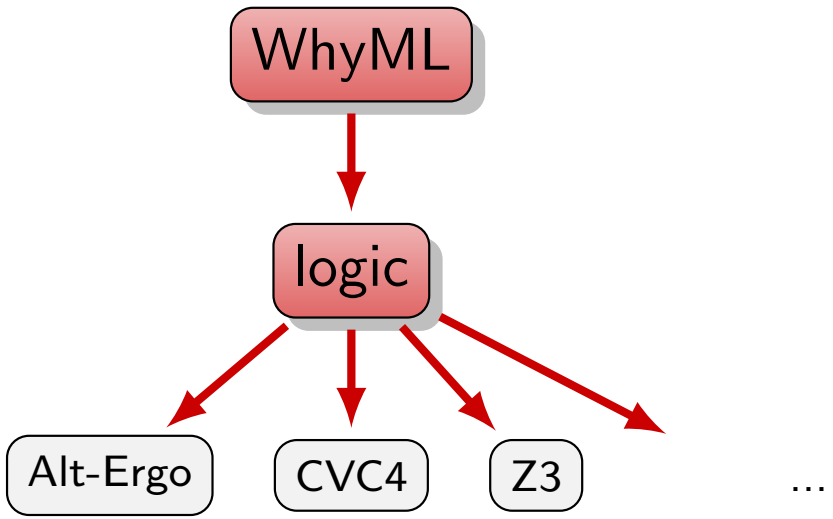
un programme qui est une preuve

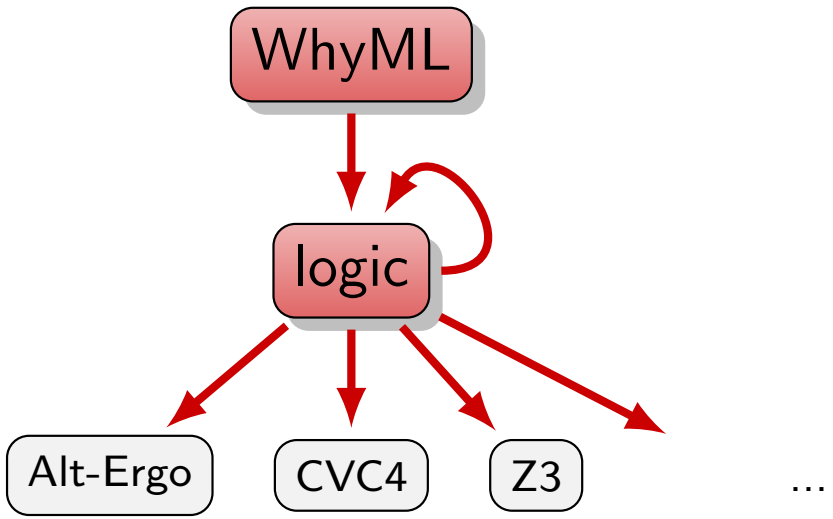
$f \leftarrow 1$
`for` $i = 1$ `to` n `do`
  `invariant` $f = (i-1)! \wedge f \geq 1$
  $f \leftarrow i \times f$
`assert` $f = n!$
`assert` $n! \geq 1$

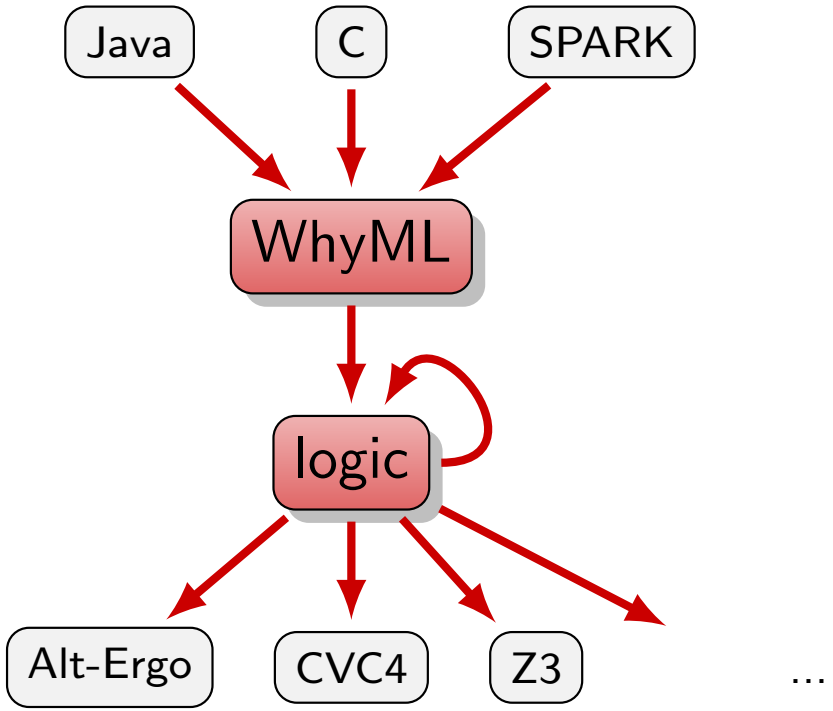
- tout le programme est fantôme
  (on ne va pas l'exécuter)

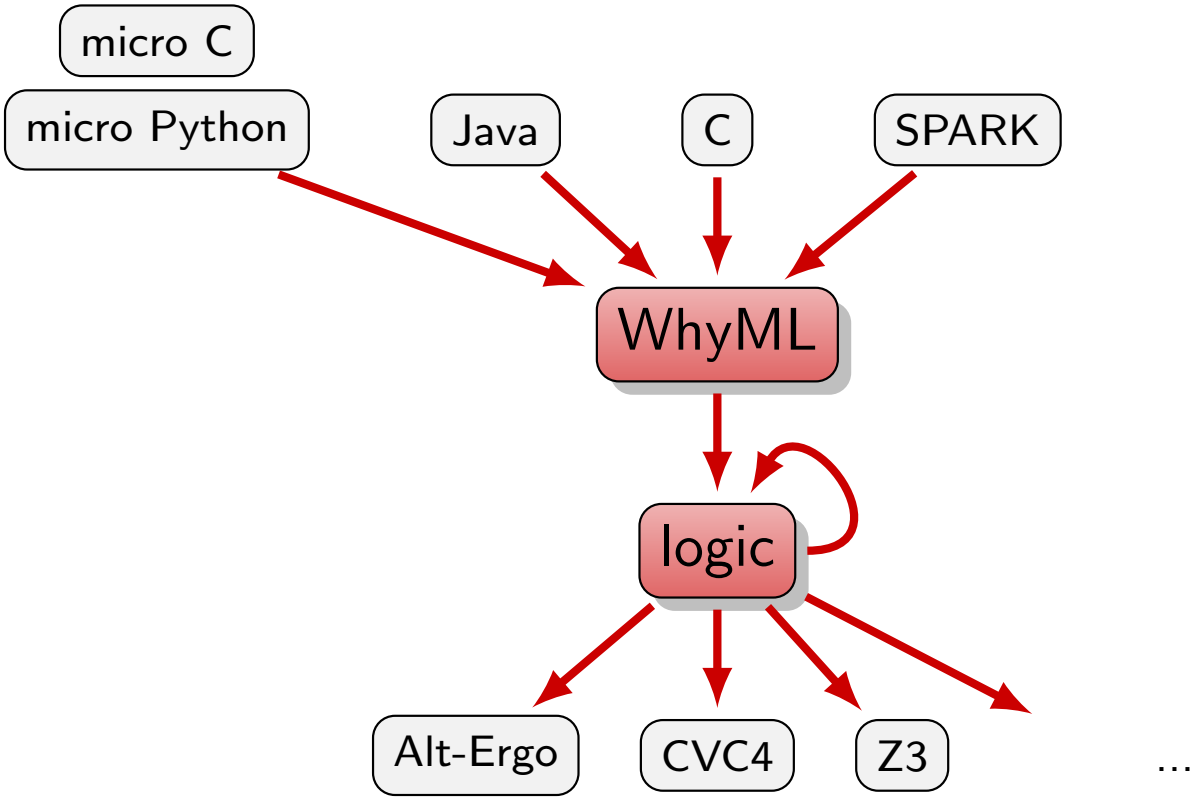
- on a fait une preuve par récurrence
  (ce qu'un démonstrateur ne fait pas par lui-même)

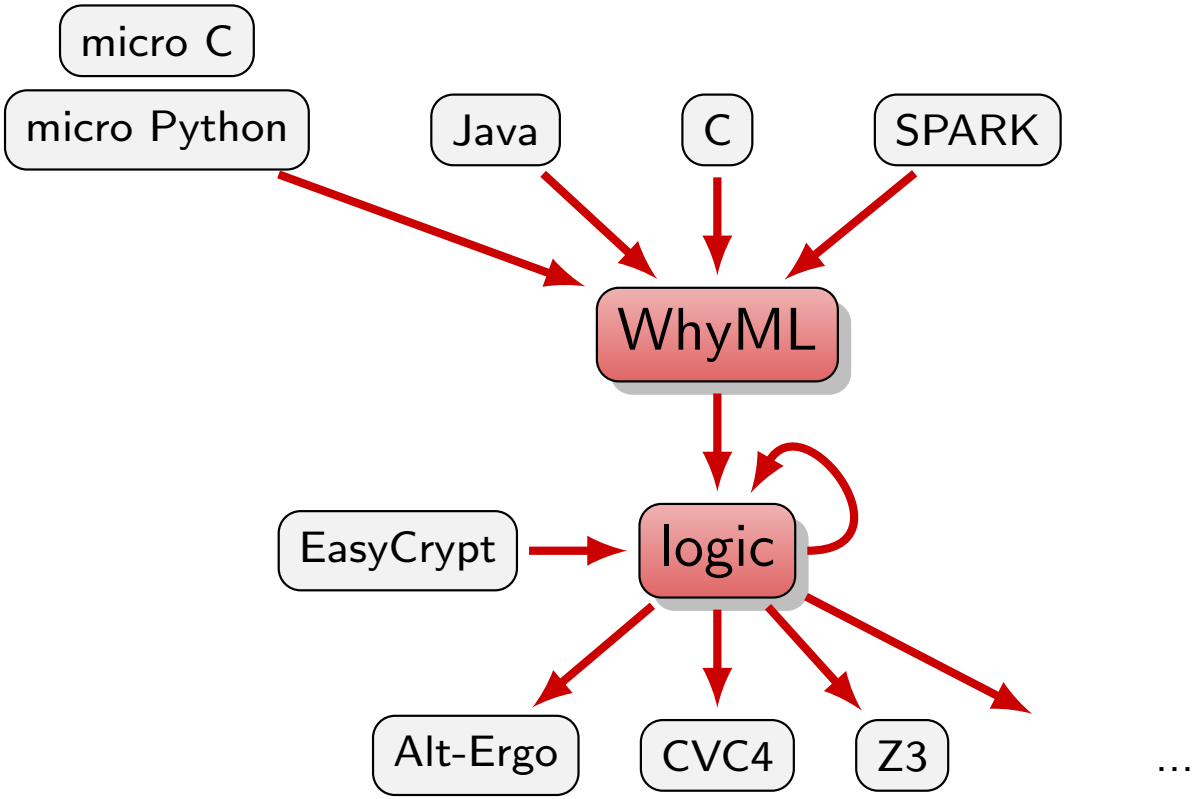
et bien d'autres choses encore

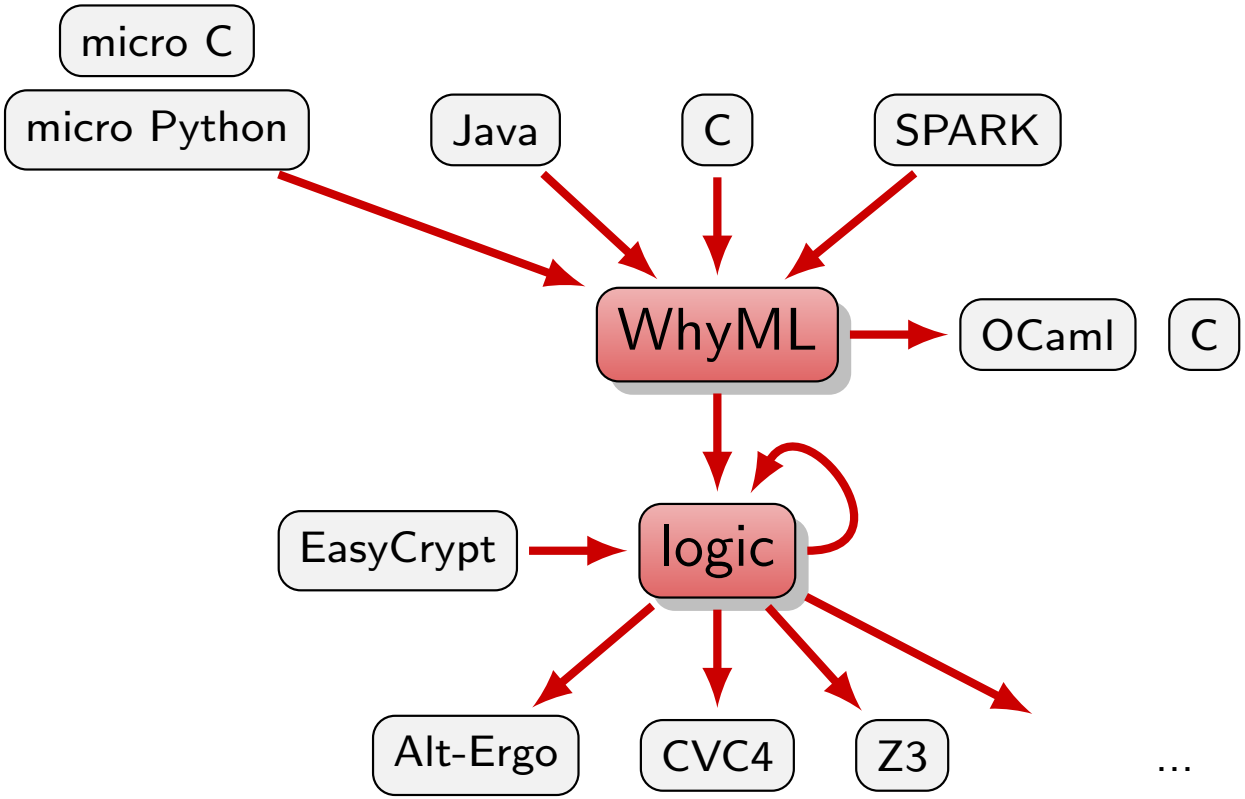

WhyML
logic
Alt-Ergo
CVC4
Z3
...

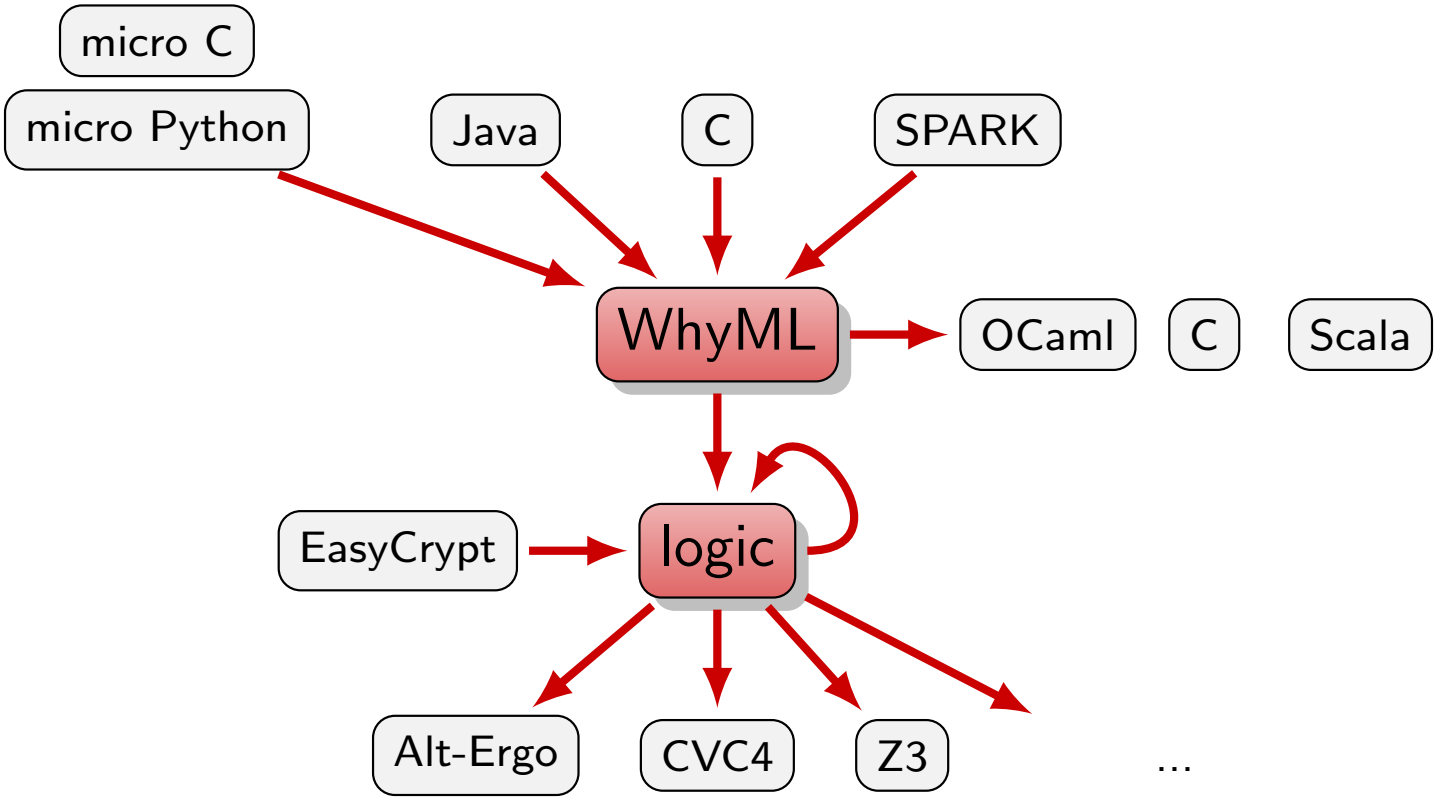

WhyML
logic
Alt-Ergo
CVC4
Z3
...


Java
C
SPARK
WhyML
logic
Alt-Ergo
CVC4
Z3
...


micro C
micro Python
Java
C
SPARK
WhyML
logic
Alt-Ergo
CVC4
Z3
...


micro C
micro Python
Java
C
SPARK
WhyML
EasyCrypt
logic
Alt-Ergo
CVC4
Z3
...


micro C
micro Python
Java
C
SPARK
WhyML
OCaml
C
EasyCrypt
logic
Alt-Ergo
CVC4
Z3
...


micro C
micro Python
Java
C
SPARK
WhyML
OCaml
C
Scala
EasyCrypt
logic
Alt-Ergo
CVC4
Z3
...

plus d'infos sur `http://why3.lri.fr/`

- documentation
- publications
- galerie de près de 200 exemples
- notes de cours
- projets utilisant Why3

conclusion

à emporter

- on peut vérifier un programme, une fois pour toutes
- on a des outils pour le faire,
  et en particulier des démonstrateurs automatiques
- cela reste très coûteux en moyens humains
- le langage de programmation importe peu
- un programme peut être une preuve
- aller voir The Princess Bride si vous ne connaissez pas