Vérification déductive de programmes avec l'outil Why3

Jean-Christophe Filliâtre
CNRS

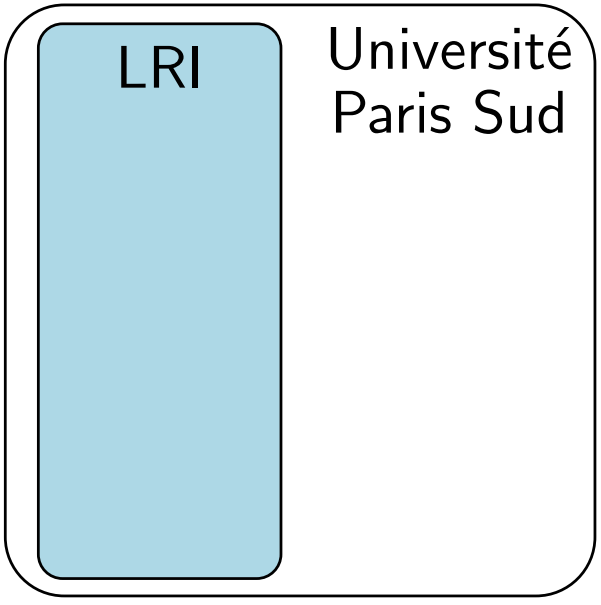
Séminaire SIESTE
ENS Lyon

2 octobre 2018

l'équipe VALS — http://vals.lri.fr/

UNIVERSITÉ PARIS SUD
université PARIS-SACLAY

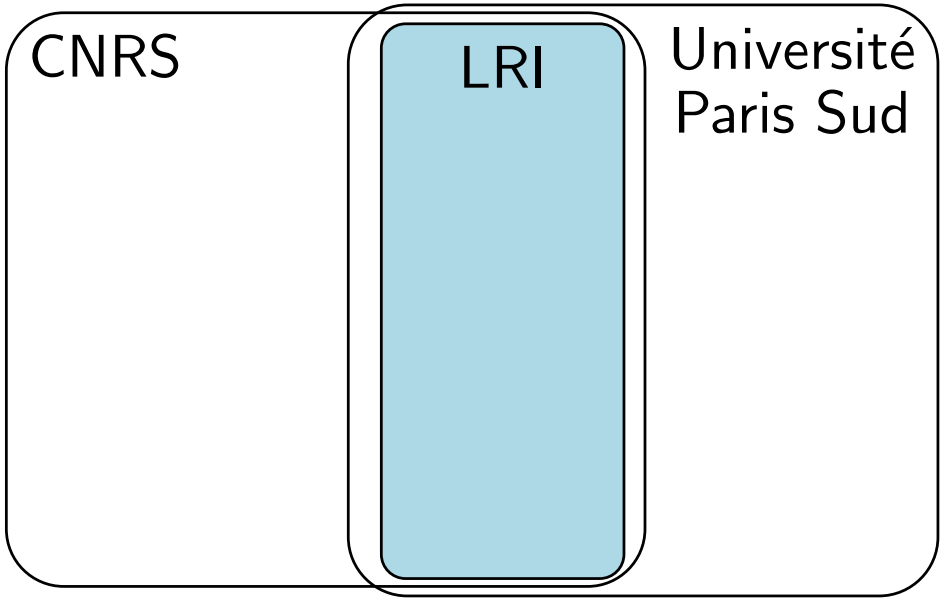
l'équipe VALS — http://vals.lri.fr/



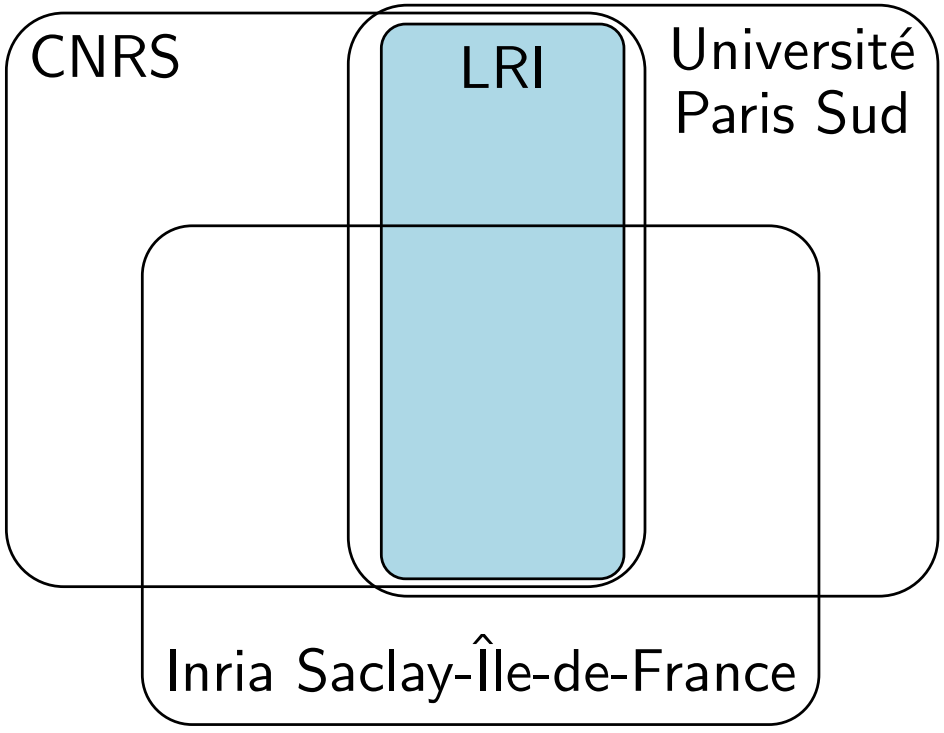
UNIVERSITÉ PARIS SUD
université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

l'équipe VALS — http://vals.lri.fr/



l'équipe VALS — http://vals.lri.fr/



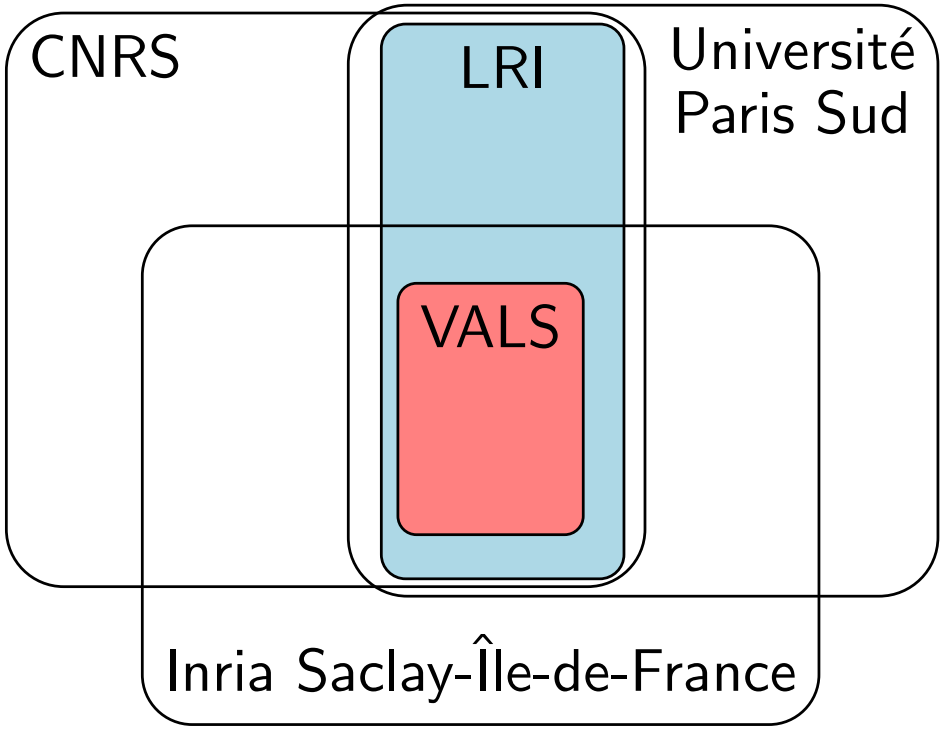
UNIVERSITÉ PARIS SUD
université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

cnrs

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY - ÎLE-DE-FRANCE

l'équipe VALS — http://vals.lri.fr/



UNIVERSITÉ PARIS SUD
université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

cnrs

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY - ÎLE-DE-FRANCE

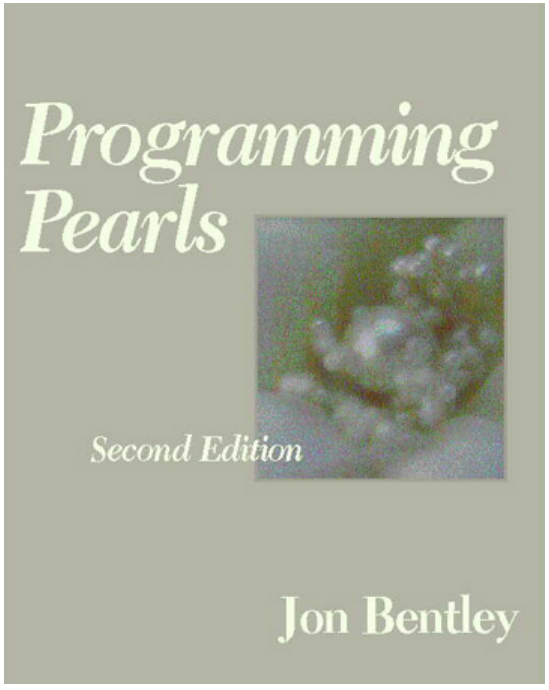
Software is hard. – Don Knuth

pourquoi ?

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

un exemple célèbre : *binary search*

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

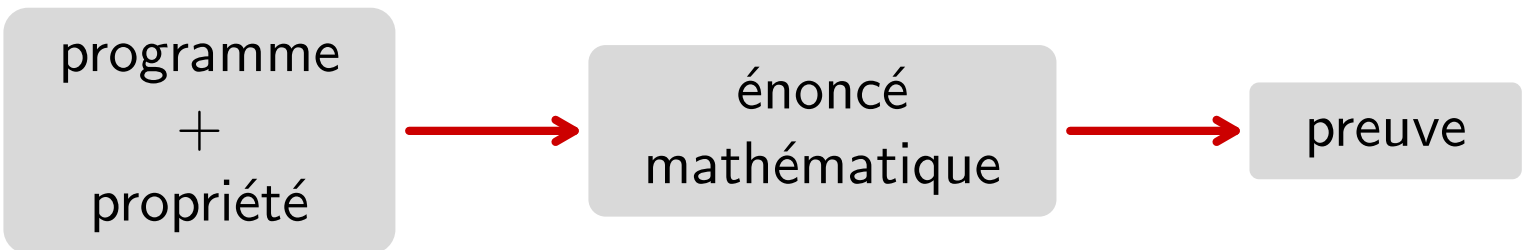
que souhaite-t-on prouver ?

- 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

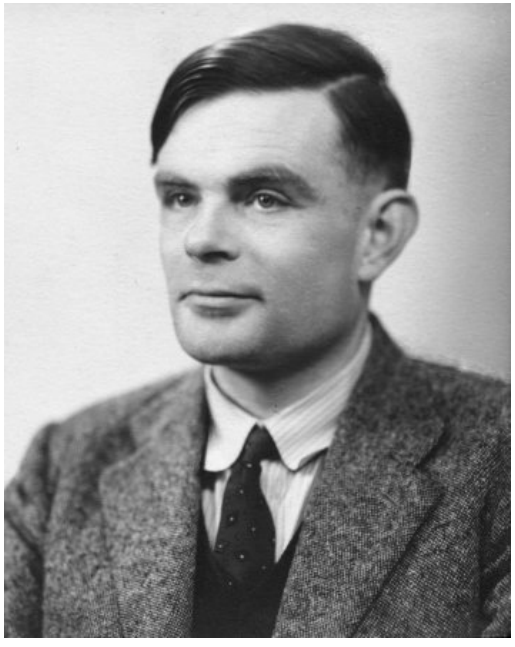
- correction fonctionnelle
  - le programme fait ce qu'il est censé faire

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

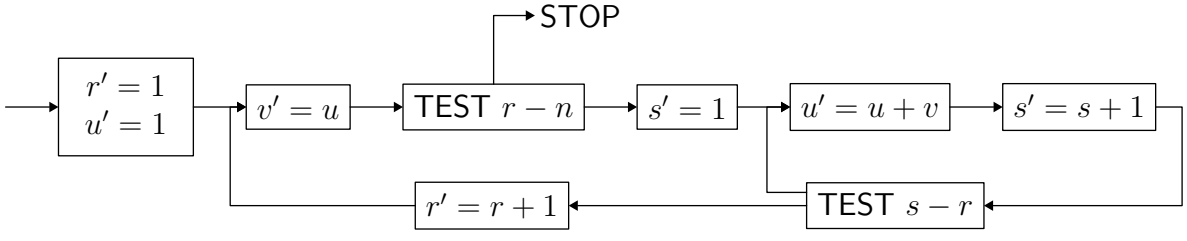
cet exposé présente la vérification déductive



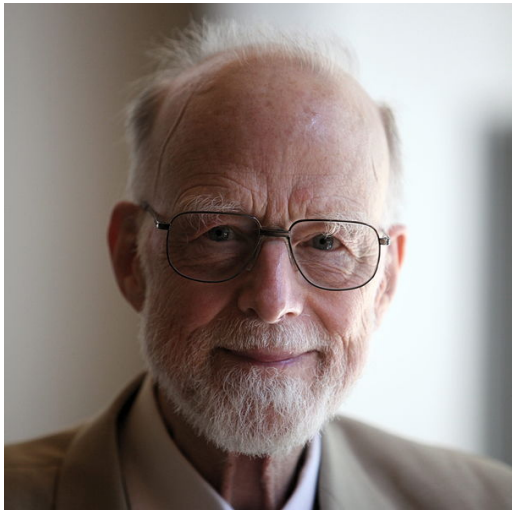
ce n'est pas nouveau



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

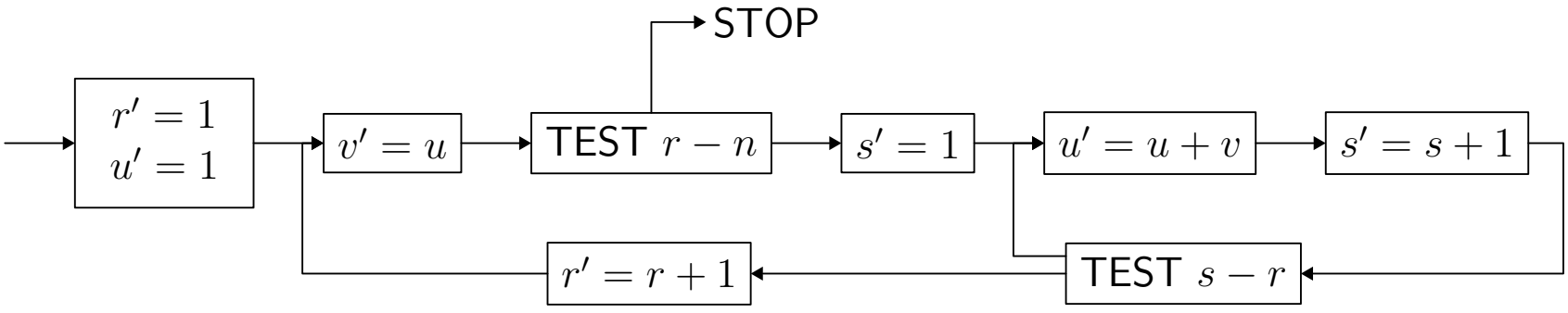


ce n'est pas nouveau

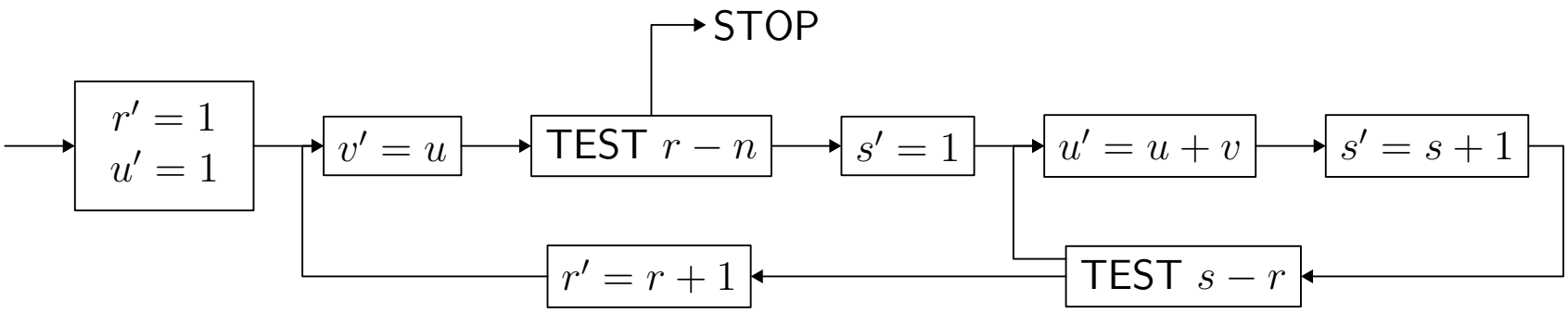


Tony Hoare.
An Axiomatic Basis for Computer Programming.
1969.

checking a large routine (Turing, 1949)

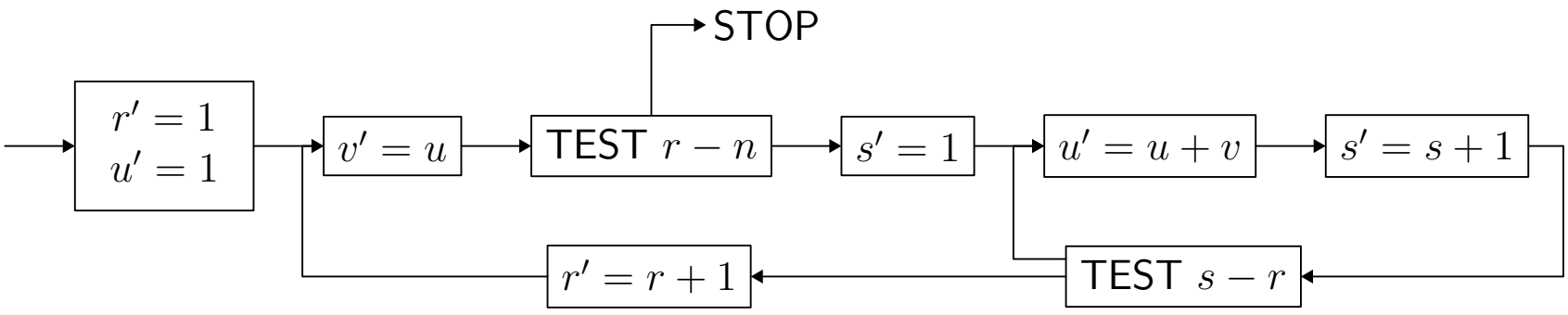


checking a large routine (Turing, 1949)



$u \leftarrow 1$
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**

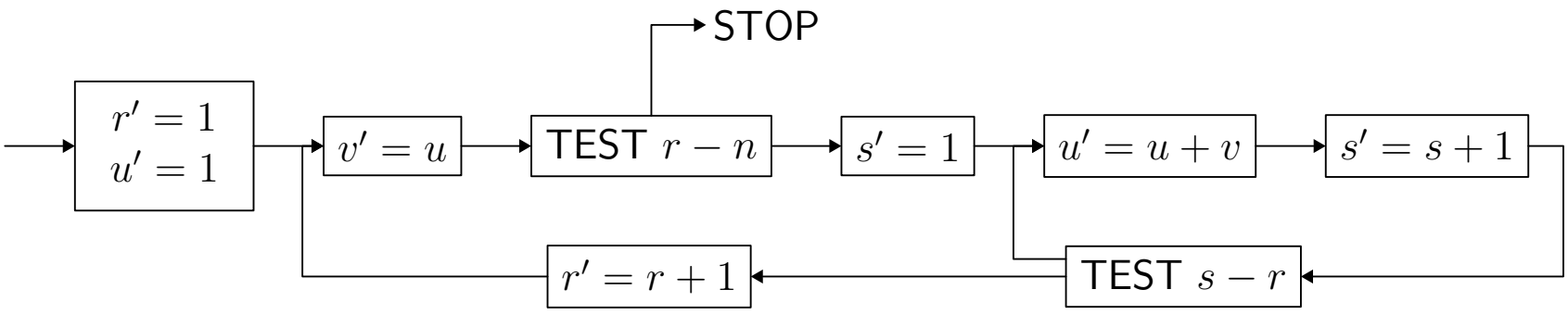
  $v \leftarrow u$

  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

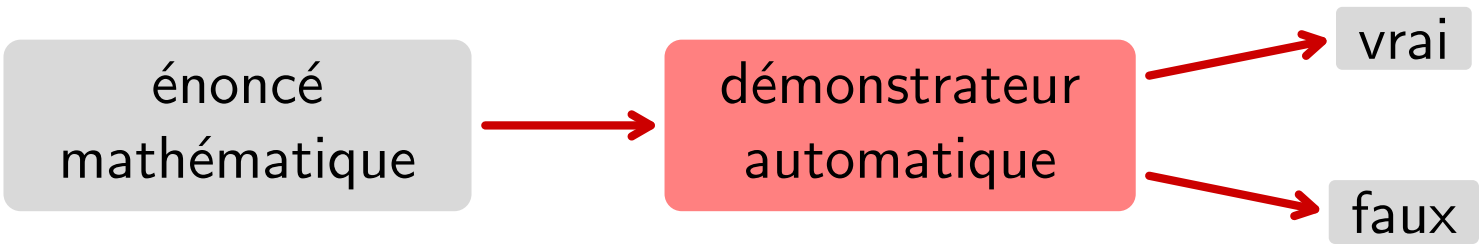
```
forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = n!) /\
  (0 <= n - 1 ->
    1 = 0! /\
    (forall u:int.
      (forall r:int. 0 <= r /\ r <= n - 1 -> u = r! ->
        (1 > r -> u = (r + 1)!) /\
        (1 <= r ->
          u = 1 * r! /\
          (forall u1:int.
            (forall s:int. 1 <= s /\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

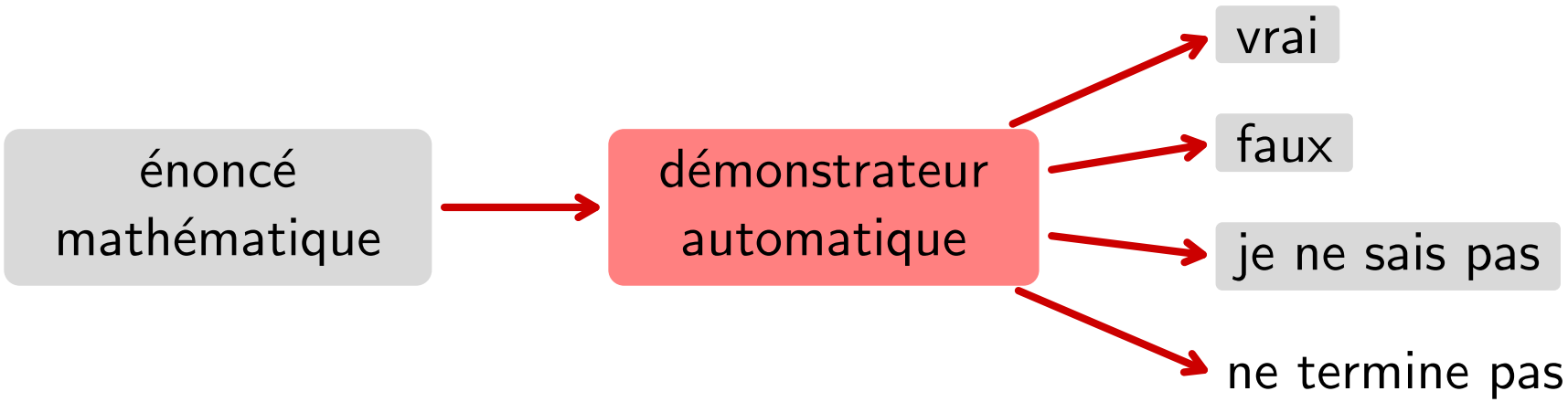


il n'est pas possible d'écrire un tel programme
(Turing/Church, 1936, d'après Gödel)

c'est le théorème anti-chômage pour les mathématiciens

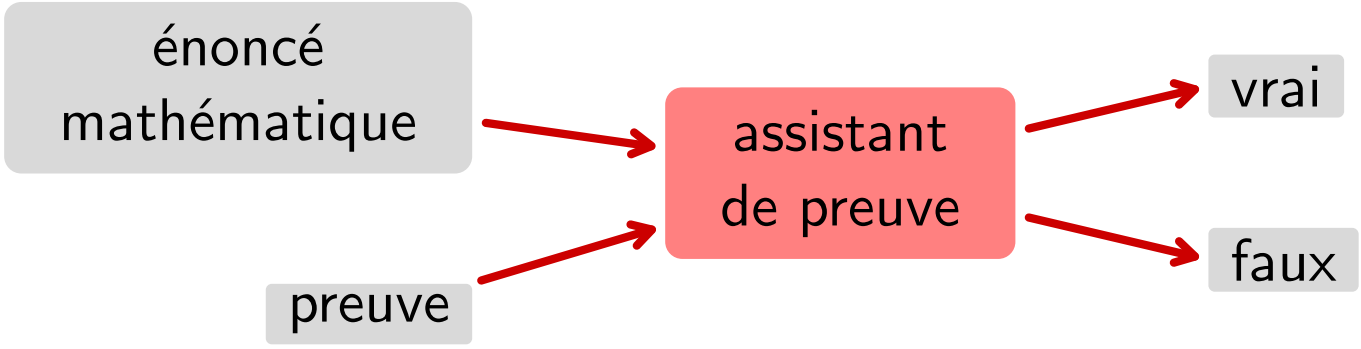
Kurt Gödel

démonstration automatique



exemples : Z3, CVC4, Alt-Ergo, Vampire, E prover, etc.

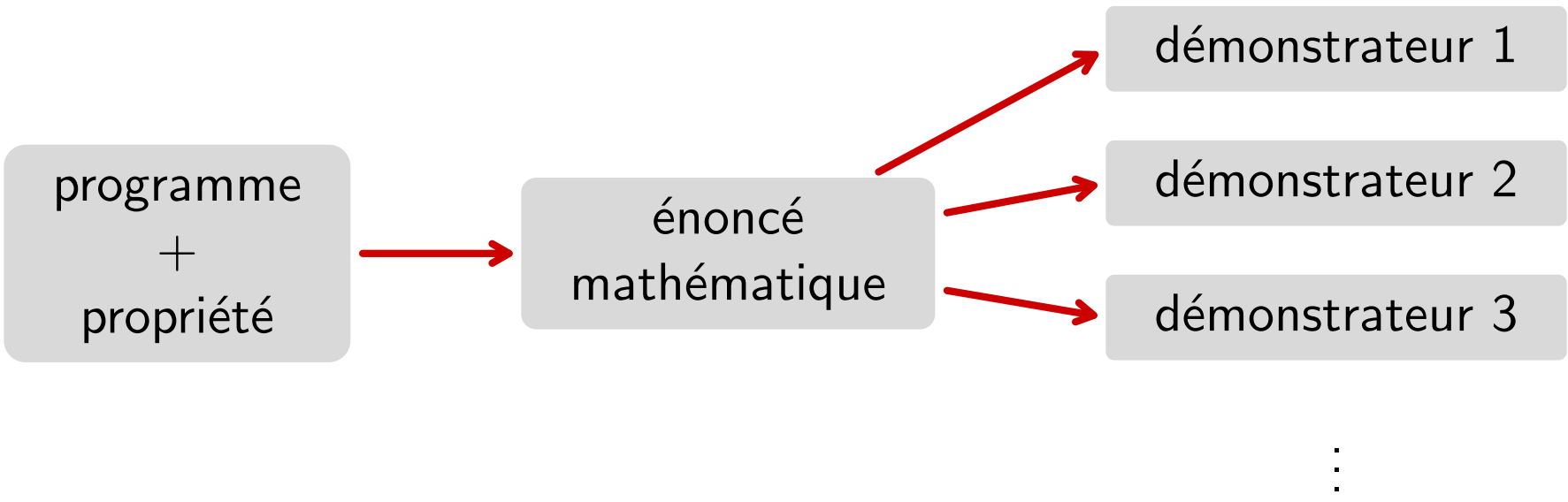
si on se contente de vérifier une preuve, cela redevient décidable



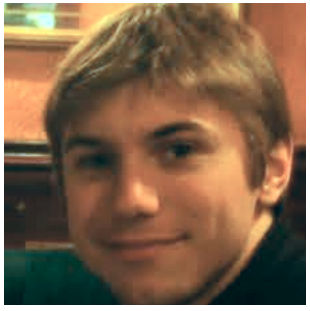
exemples : Coq, Isabelle, PVS, HOL-light, etc.

Why3, un outil de vérification déductive

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



en collaboration avec

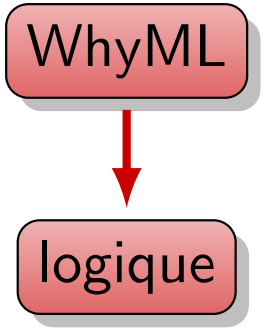


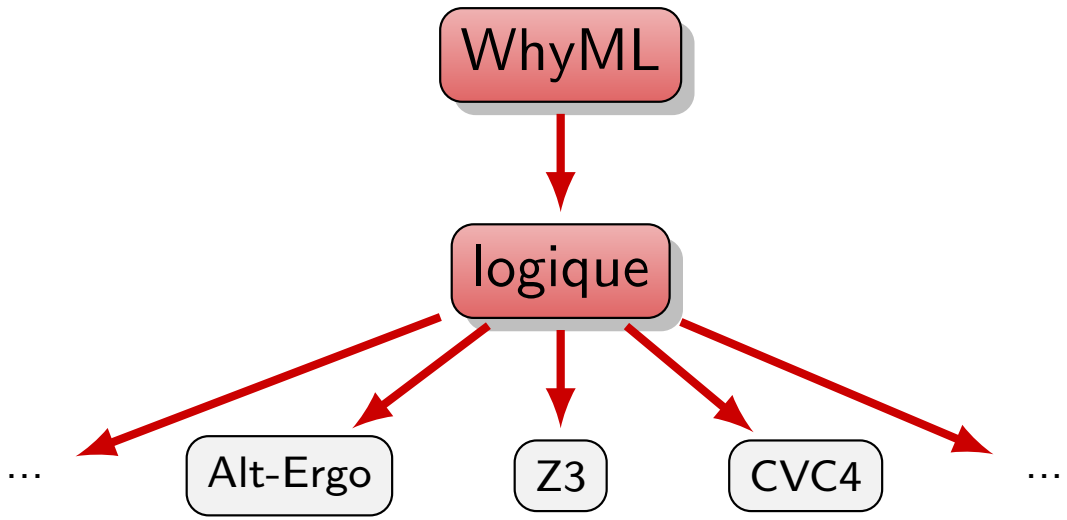
François Bobot

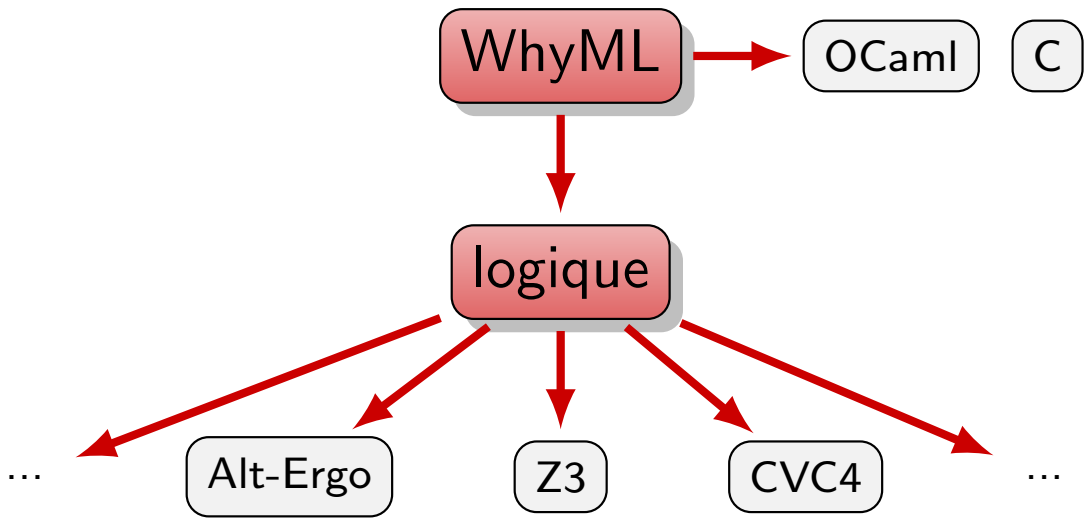
Claude Marché

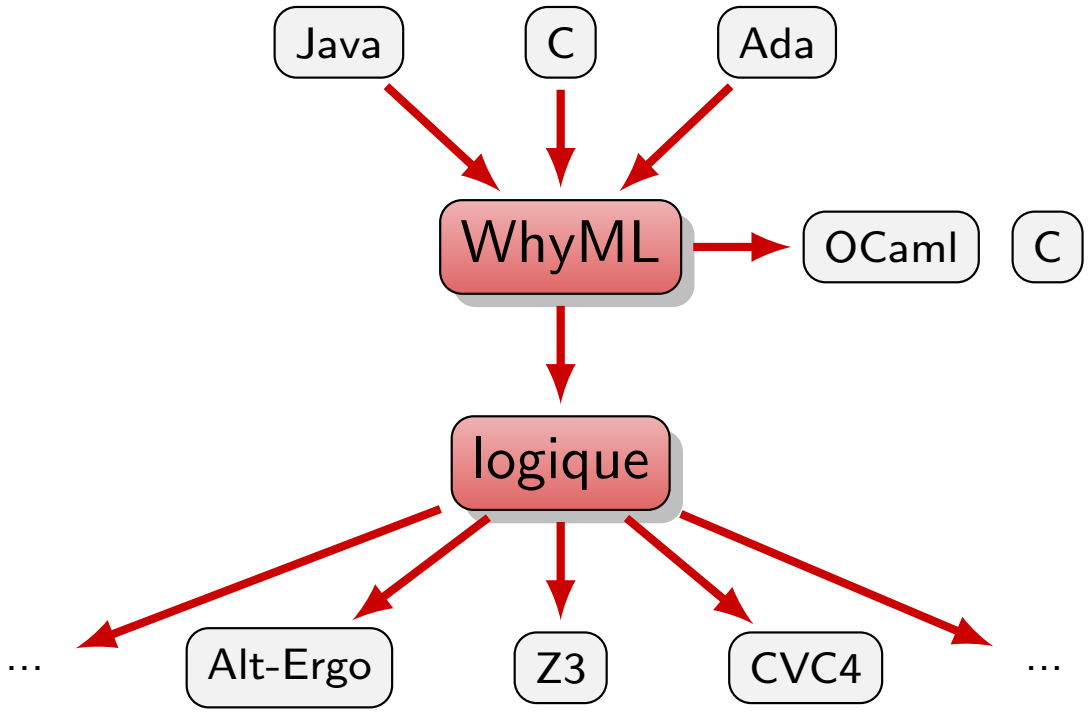
Guillaume Melquiond

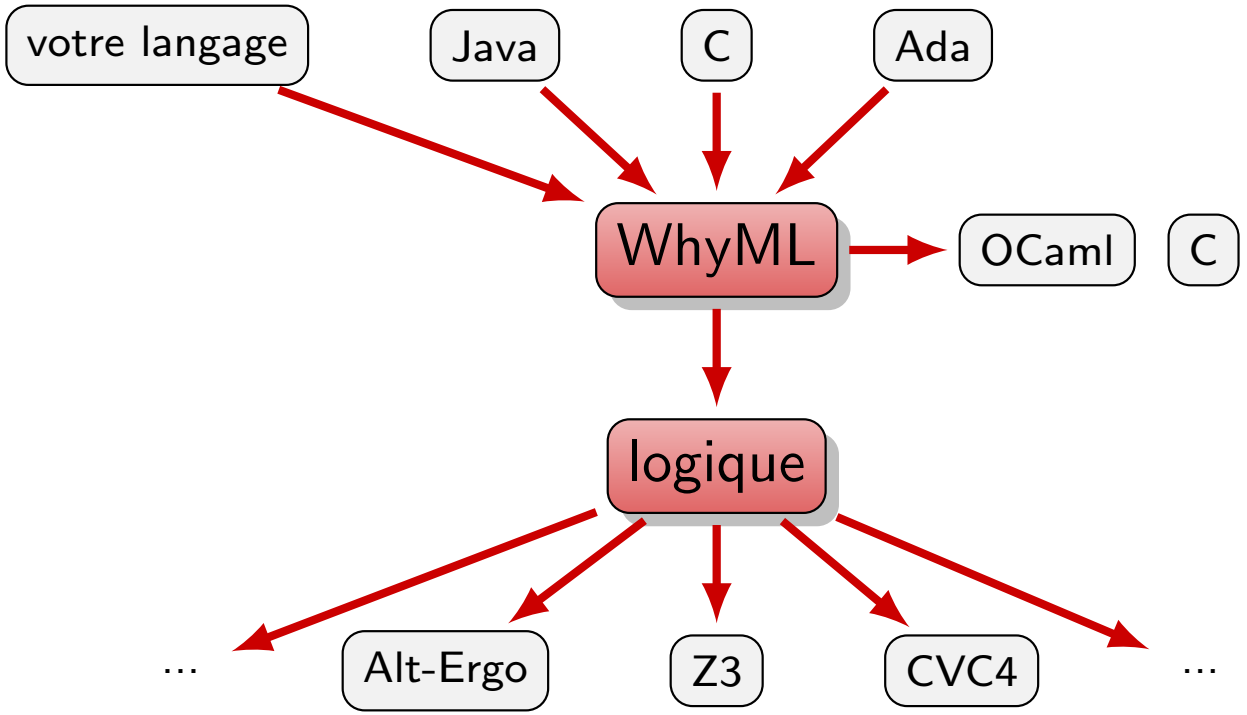
Andrei Paskevich


WhyML
logique


WhyML
logique
...
Alt-Ergo
Z3
CVC4
...


WhyML
OCaml
C
logique
...
Alt-Ergo
Z3
CVC4
...


Java
C
Ada
WhyML
OCaml
C
logique
...
Alt-Ergo
Z3
CVC4
...


votre langage
Java
C
Ada
WhyML
OCaml
C
logique
...
Alt-Ergo
Z3
CVC4
...

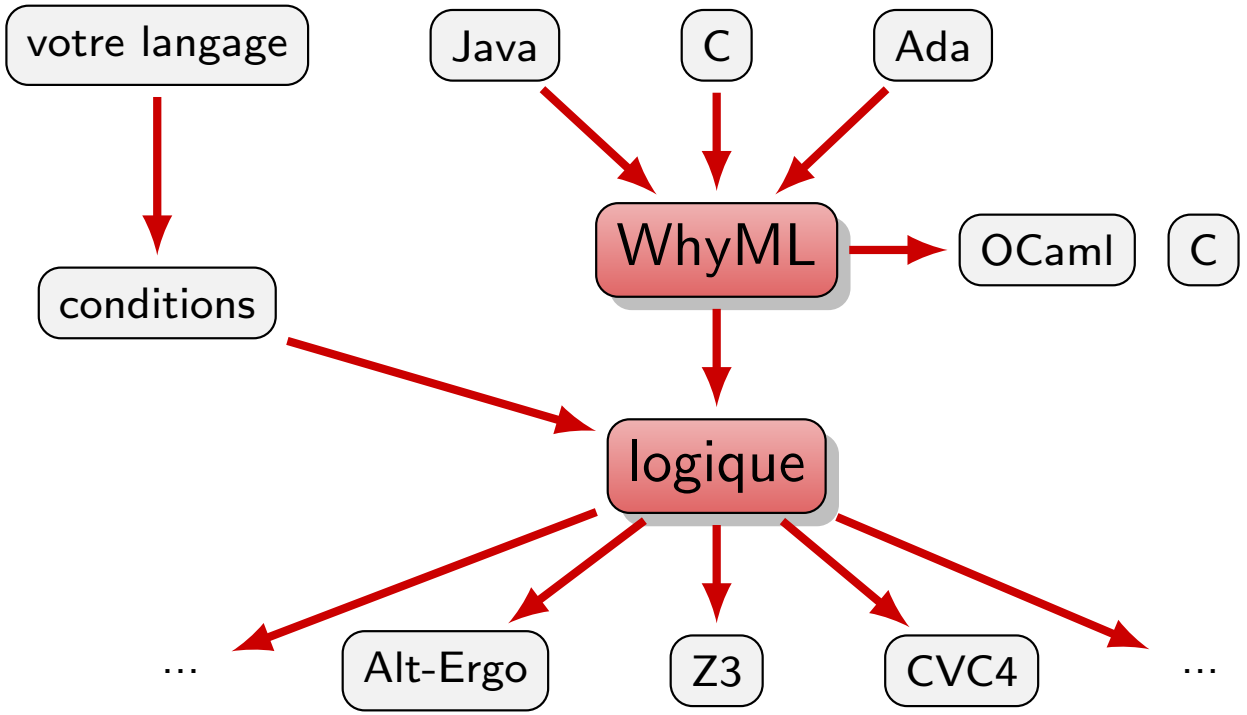
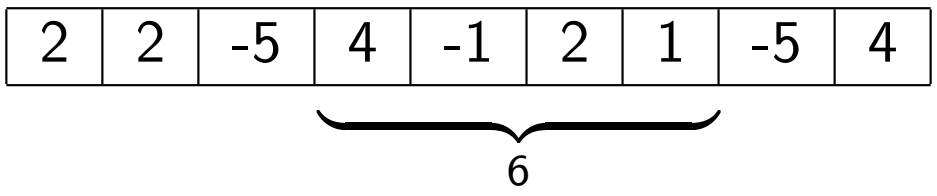

votre langage
Java
C
Ada
WhyML
OCaml
C
conditions
logique
...
Alt-Ergo
Z3
CVC4
...

illustration sur un exemple

sous-tableau de somme maximale

étant donné un tableau d'entiers,
déterminer la somme maximale parmi les sous-tableaux

exemple :

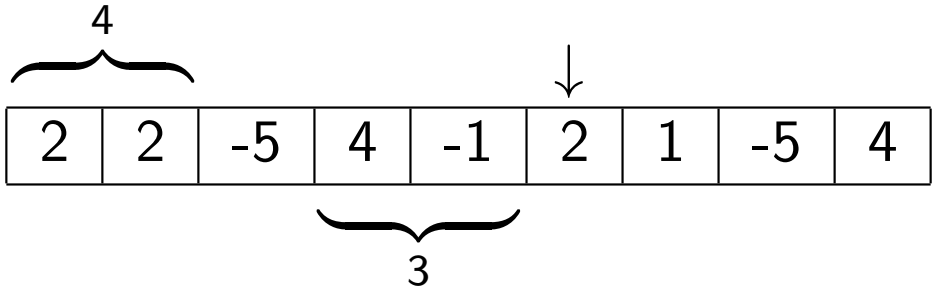


note : un sous-tableau peut être vide, et maximal quand tous les éléments sont négatifs

l'algorithme de Kadane

une solution linéaire, en espace constant, due à Jay Kadane (1977)
(l'histoire est racontée dans *Programming Pearls* de Jon Bentley)

principe : parcourir le tableau et maintenir à la fois le maximum et le suffixe maximum



à propos du code fantôme

code fantôme

du code ajouté au programme
pour faciliter la spécification et/ou la preuve

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

conséquence : le code fantôme peut être effacé sans modification observable

$$
\begin{array}{rcll}
t & ::= & c & \textit{constante} \\
 & | & x & \textit{variable} \\
 & | & \lambda x.\, t & \textit{fonction} \\
 & | & t\ t & \textit{application} \\
 & | & !r & \textit{accès} \\
 & | & r := t & \textit{affectation} \\
 & | & \texttt{ghost}\ t & \textit{code fantôme}
\end{array}
$$

$$\vdash \; t \; : \; \tau$$

$$\vdash \; t \; : \; \tau, \; \beta, \; \varepsilon$$

où

- $\beta \in \{\top, \bot\}$ indique le statut fantôme
- $\varepsilon \in \{\top, \bot\}$ indique la présence d'un effet non fantôme

$$\vdash \; t \; : \; \tau, \; \beta, \; \varepsilon$$

où

- $\beta \in \{\top, \bot\}$ indique le statut fantôme
- $\varepsilon \in \{\top, \bot\}$ indique la présence d'un effet non fantôme

condition systématique

$$\beta = \top \; \Rightarrow \; \varepsilon = \bot$$

$$\vdash \; t \; : \; (\tau_2 \overset{\beta_2}{\xrightarrow{\varepsilon_1}} \tau_1), \; \beta, \; \varepsilon$$

où

- β_2 indique le statut fantôme de l'argument
- ε_1 est l'effet *latent* de la fonction non fantôme

typage d'une application

deux cas de figure pour typer l'application $(t_1\ t_2)$:

- $t_1 : \tau_2^{\bot} \xrightarrow{\varepsilon_1} \tau_1$ fonction avec paramètre régulier
- $t_1 : \tau_2^{\top} \xrightarrow{\varepsilon_1} \tau_1$ fonction avec paramètre fantôme

$$\frac{\vdash\ t_1\ :\ \tau_2^{\perp} \xrightarrow{\varepsilon_1} \tau_1,\ \beta_1,\ \varepsilon_2 \qquad \vdash\ t_2\ :\ \tau_2,\ \beta_2,\ \varepsilon_3}{\vdash\ (t_1\ t_2)\ :\ \tau_1,\ \beta_1 \vee \beta_2,\ \varepsilon_1 \vee \varepsilon_2 \vee \varepsilon_3}$$

le statut de t_2 contamine le statut de l'application

$$(\lambda v^{\bot} : \mathtt{int}.\, g := v;\ !r)\ 42$$

$$\frac{\vdash\ t_1\ :\ \mathtt{int}^{\bot} \xrightarrow{\bot} \mathtt{int},\ \bot,\ \bot \qquad \vdash\ t_2\ :\ \mathtt{int},\ \bot,\ \bot}{\vdash\ (t_1\ t_2)\ :\ \mathtt{int},\ \bot,\ \bot}$$

$$(\lambda v^{\bot} : \mathtt{int}.\, g := v;\ !r)\ (\text{ghost } 42)$$

$$\frac{\vdash\ t_1\ :\ \mathtt{int}^{\bot} \xrightarrow{\bot} \mathtt{int},\ \bot,\ \bot \qquad \vdash\ t_2\ :\ \mathtt{int},\ \top,\ \bot}{\vdash\ (t_1\ t_2)\ :\ \mathtt{int},\ \top,\ \bot}$$

$$(\lambda v^{\bot} : \mathtt{int}.\, g := v;\ r := 0)\ (\text{ghost } 42)$$

est rejeté, car

$$\frac{\vdash\ t_1\ :\ \mathtt{int}^{\bot} \xrightarrow{\top} \mathtt{unit},\ \bot,\ \bot \qquad \vdash\ t_2\ :\ \mathtt{int},\ \top,\ \bot}{\vdash\ (t_1\ t_2)\ :\ \mathtt{unit},\ \top,\ \top}$$

viole la condition $\beta = \top \Rightarrow \varepsilon = \bot$

on peut accepter l'exemple précédent

$(\lambda v^{\bot} : \texttt{int}.\, g := v;\ r := 0)\ (\text{ghost } 42)$

si on change le statut fantôme du paramètre

$(\lambda v^{\top} : \texttt{int}.\, g := v;\ r := 0)\ (\text{ghost } 42)$

application avec paramètre fantôme

$$\frac{\vdash\ t_1\ :\ \tau_2^{\top} \xrightarrow{\varepsilon_1} \tau_1,\ \beta_1,\ \varepsilon_2 \qquad \vdash\ t_2\ :\ \tau_2,\ \beta_2,\ \bot}{\vdash\ (t_1\ t_2)\ :\ \tau_1,\ \beta_1,\ \varepsilon_1 \vee \varepsilon_2}$$

- l'argument t_2 doit être sans effet
- le statut fantôme de l'application est celui de t_1

The Spirit of Ghost Code

J-C F., Léon Gondelman, Andrei Paskevich

CAV 2014

`http://hal.inria.fr/hal-00873187`

conclusion

l'outil Why3

http://why3.lri.fr/

- logiciel libre
- plus de 150 programmes prouvés
- documentation, notes de cours (y compris en français)

- la vérification déductive est une méthode formelle de preuve de programme (ce n'est pas la seule)
- elle s'appuie en particulier sur les démonstrateurs, automatiques et interactifs, qui mécanisent les raisonnements logiques
- cela reste un processus très coûteux, notamment en moyens humains (écrire des spécifications, des invariants, des preuves)

défis scientifiques

- définir de meilleurs langages de programmation, mieux adaptés à la preuve
- définir de meilleurs langages logiques, plus expressifs
- définir de meilleurs démonstrateurs automatiques

} tension

des questions ?