Vérification déductive de programmes

Jean-Christophe Filliâtre
CNRS

Diplôme Universitaire
spécialité Numérique et Sciences Informatiques

20 juin 2019

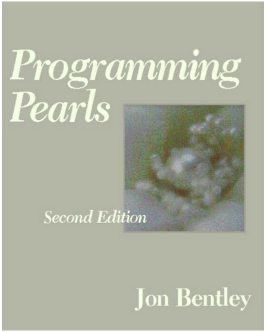
Software is hard. – Don Knuth

pourquoi ?

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

un exemple célèbre : la recherche dichotomique

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...

et pourtant

en 2006, un bug a été trouvé dans le code 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.

une autre réponse

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)

encore une autre réponse

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

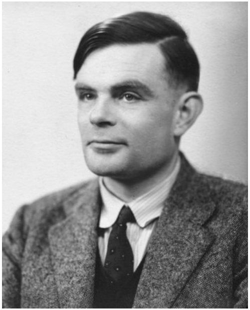
de nombreuses approches

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

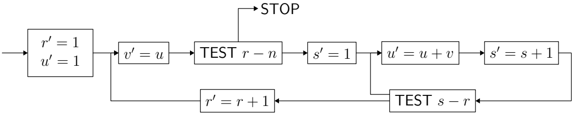
cet exposé présente la vérification déductive

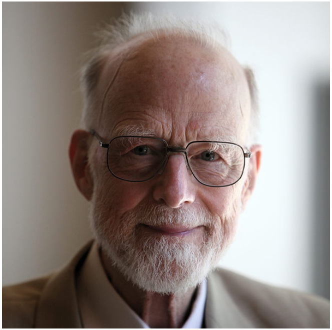
programme
$+$
propriété
$\longrightarrow$ énoncé mathématique $\longrightarrow$ preuve

ce n’est pas nouveau



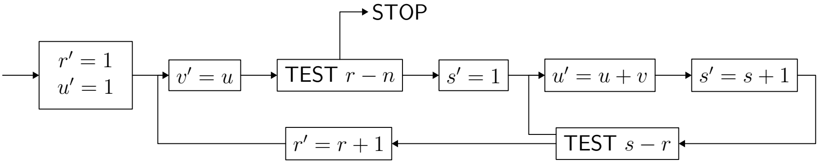
A. M. Turing. Checking a large routine. 1949.



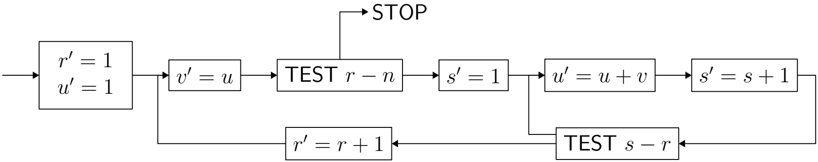


Tony Hoare.
An Axiomatic Basis for Computer Programming. 1969.

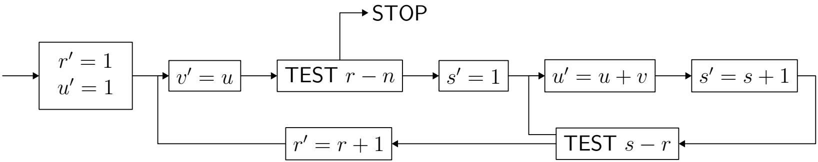
checking a large routine (Turing, 1949)



checking a large routine (Turing, 1949)



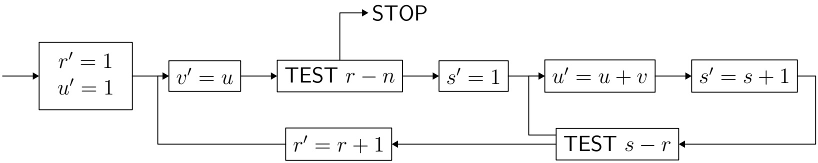

```
u = 1
for r in range(0, n):
  v = u
  for s in range(1, r + 1):
    u = u + v
```



précondition $\{\mathtt{n} \geq 0\}$

```
u = 1
for r in range(0, n):
  v = u
  for s in range(1, r + 1):
    u = u + v
```

postcondition $\{\mathtt{u} = \textit{fact}(\mathtt{n})\}$



précondition $\{\mathtt{n} \geq 0\}$

```
u = 1
for r in range(0, n):        invariant {u = fact(r)}
  v = u
  for s in range(1, r + 1):    invariant {u = s × fact(r)}
    u = u + v
```

postcondition $\{\mathtt{u} = \mathit{fact}(\mathtt{n})\}$

axiome $fact(0) = 1$
axiome $\forall n.\ n \geq 1 \Rightarrow fact(n) = n \times fact(n-1)$

$$
\begin{array}{l}
\forall n.\ n \geq 0 \Rightarrow \\
\quad (0 > n-1 \Rightarrow 1 = fact(n)) \wedge \\
\quad (0 <= n-1 \Rightarrow \\
\qquad 1 = fact(0) \wedge \\
\qquad (\forall u. \\
\qquad\quad (\forall r.\ 0 \leq r\ \wedge\ r \leq n-1 \Rightarrow u = fact(r) \Rightarrow \\
\qquad\qquad (1 > r \Rightarrow u = fact(r+1)) \wedge \\
\qquad\qquad (1 \leq r \Rightarrow \\
\qquad\qquad\quad u = 1 \times fact(r) \wedge \\
\qquad\qquad\quad (\forall u_1. \\
\qquad\qquad\qquad (\forall s.\ 1 \leq s\ \wedge\ s \leq r \Rightarrow u_1 = s \times fact(r) \Rightarrow \\
\qquad\qquad\qquad\quad (\forall u_2. \\
\qquad\qquad\qquad\qquad u_2 = u_1 + u \Rightarrow u_2 = (s+1) \times fact(r))) \wedge \\
\qquad\qquad\qquad (u_1 = (r+1) \times fact(r) \Rightarrow u_1 = fact(r+1))))) \wedge \\
\qquad\quad (u = fact((n-1)+1) \Rightarrow u = fact(n))))
\end{array}
$$

c'est un énoncé logique qui exprime

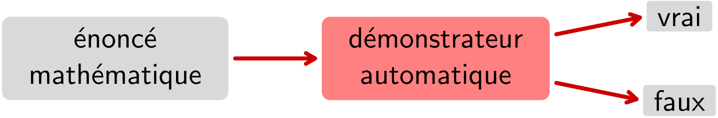
- la sûreté
  - pas de division par zéro
  - accès dans les bornes des tableaux
  - terminaison
- respect des spécifications
  - les invariants sont initialisés et préservés
  - les postconditions sont établies dans les fonctions
  - les préconditions sont établies dans les appels

ensuite

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

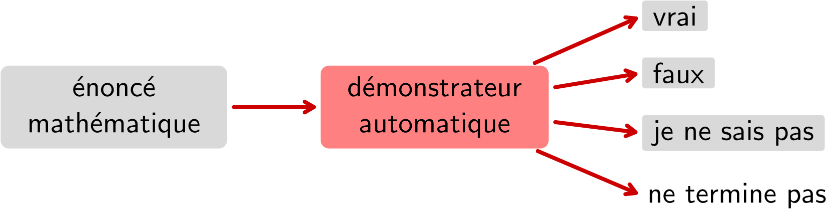
aussi, on se tourne vers des outils qui mécanisent le raisonnement mathématique


énoncé
mathématique
démonstrateur
automatique
vrai
faux

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

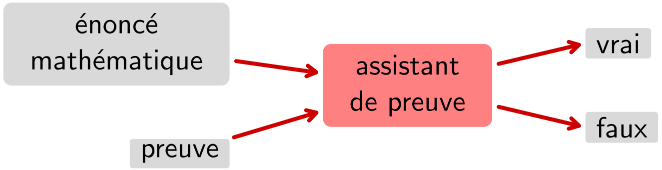
c'est le théorème anti-chômage pour les mathématiciens

Kurt Gödel



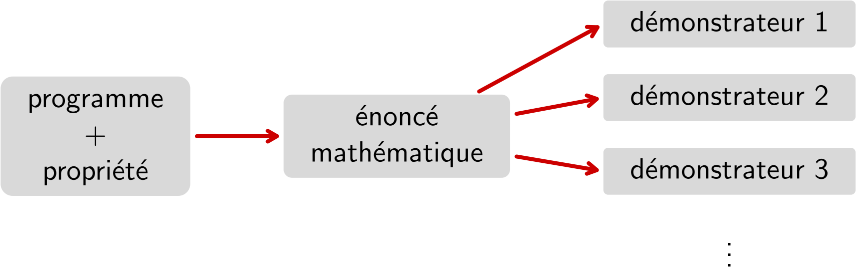
exemples : Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

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



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

Why3, un outil de vérification déductive



démo

calculer la condition de vérification

$x \leftarrow x + 1$ « x est pair »

par récurrence sur le programme

« $x + 1$ est pair » $x \leftarrow x + 1$ « x est pair »

$$\ll x + 1 \text{ est pair} \gg \qquad x \leftarrow x + 1 \qquad \ll x \text{ est pair} \gg$$

$$\ll \psi[e] \gg \qquad x \leftarrow e \qquad \ll \psi[x] \gg$$

$$\ll x + 1 \text{ est pair} \gg \qquad x \leftarrow x + 1 \qquad \ll x \text{ est pair} \gg$$

$$\ll \psi[e] \gg \qquad x \leftarrow e \qquad \ll \psi[x] \gg$$

$$\begin{array}{lll} \texttt{if } c \texttt{ then} & P_1 & \ll \psi \gg \\ \qquad\;\; \texttt{else} & P_2 & \end{array}$$

$\ll x + 1 \text{ est pair} \gg \qquad x \leftarrow x + 1 \qquad \ll x \text{ est pair} \gg$

$\ll \psi[e] \gg \qquad x \leftarrow e \qquad \ll \psi[x] \gg$

$$\begin{array}{lll} \texttt{if } c \texttt{ then} & P_1\, \psi & \ll \psi \gg \\ \phantom{\texttt{if } c} \texttt{ else} & P_2\, \psi & \end{array}$$

par récurrence sur le programme

$$\ll x + 1 \text{ est pair} \gg \qquad x \leftarrow x + 1 \qquad \ll x \text{ est pair} \gg$$

$$\ll \psi[e] \gg \qquad x \leftarrow e \qquad \ll \psi[x] \gg$$

$$\begin{array}{ll} \texttt{if } c \texttt{ then} & \varphi_1 \, P_1 \, \psi \\ \texttt{else} & \varphi_2 \, P_2 \, \psi \end{array} \qquad \ll \psi \gg$$

« $x + 1$ est pair »	$x \leftarrow x + 1$	« x est pair »
« $\psi[e]$ »	$x \leftarrow e$	« $\psi[x]$ »
« si c alors φ_1 sinon φ_2 »	`if` c `then` $\varphi_1\, P_1\, \psi$ `else` $\varphi_2\, P_2\, \psi$	« ψ »

≪ $x + 1$ est pair ≫	$x \leftarrow x + 1$	≪ x est pair ≫
≪ $\psi[e]$ ≫	$x \leftarrow e$	≪ $\psi[x]$ ≫
≪ si c alors φ_1 sinon φ_2 ≫	`if` c `then` φ_1 P_1 ψ `else` φ_2 P_2 ψ	≪ ψ ≫
	`if` c `then` P	≪ ψ ≫

par récurrence sur le programme

$\ll x + 1$ est pair $\gg$ $\quad x \leftarrow x + 1 \quad$ $\ll x$ est pair $\gg$

$\ll \psi[e] \gg$ $\quad x \leftarrow e \quad$ $\ll \psi[x] \gg$

$\ll$ si c alors φ_1 sinon $\varphi_2 \gg$ $\quad$ `if` c `then` $\varphi_1\, P_1\, \psi$ `else` $\varphi_2\, P_2\, \psi$ $\quad \ll \psi \gg$

`if` c `then` $\varphi\, P\, \psi$ $\quad \ll \psi \gg$

par récurrence sur le programme

« x + 1 est pair »	$x \leftarrow x + 1$	« x est pair »
« $\psi[e]$ »	$x \leftarrow e$	« $\psi[x]$ »
« si c alors φ_1 sinon φ_2 »	`if` c `then` $\varphi_1\, P_1\, \psi$ `else` $\varphi_2\, P_2\, \psi$	« ψ »
« si c alors φ sinon ψ »	`if` c `then` $\varphi\, P\, \psi$	« ψ »

$$\ll x+1 \text{ est pair} \gg \quad x \leftarrow x+1 \quad \ll x \text{ est pair} \gg$$

$$\ll \psi[e] \gg \quad x \leftarrow e \quad \ll \psi[x] \gg$$

$$\begin{array}{rl} \ll \text{si } c \text{ alors } \varphi_1 & \texttt{if } c \texttt{ then } \varphi_1\, P_1\, \psi \\ \text{sinon } \varphi_2 \gg & \qquad\quad \texttt{else } \varphi_2\, P_2\, \psi \end{array} \quad \ll \psi \gg$$

$$\begin{array}{rl} \ll \text{si } c \text{ alors } \varphi & \texttt{if } c \texttt{ then } \varphi\, P\, \psi \\ \text{sinon } \psi \gg & \end{array} \quad \ll \psi \gg$$

$$\texttt{while } c \texttt{ do } P \texttt{ done} \quad \ll \psi \gg$$

« $x + 1$ est pair »	$x \leftarrow x + 1$	« x est pair »
« $\psi[e]$ »	$x \leftarrow e$	« $\psi[x]$ »
« si c alors φ_1 sinon φ_2 »	`if` c `then` φ_1 P_1 ψ `else` φ_2 P_2 ψ	« ψ »
« si c alors φ sinon ψ »	`if` c `then` φ P ψ	« ψ »
?	`while` c `do` P `done`	« ψ »

une propriété P

- vraie initialement
- préservée par toute itération de la boucle

$$P \longrightarrow P \longrightarrow P \longrightarrow \cdots \longrightarrow P \longrightarrow P$$

en particulier, P sera vraie après la boucle
quel que soit le nombre d'itérations (y compris 0)

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			

invariant de boucle : exemple

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0

invariant de boucle : exemple

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
				0

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14		0

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
				14

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
		28		14

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
		28	1	14

invariant de boucle : exemple

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
		28	1	14
				42

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
		28	1	14
		56		42

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	b	p	q	r
7	6			
		7	6	0
		14	3	0
		28	1	14
		56	0	42

```
p = a
q = b
r = 0
while q > 0:
    if q % 2 == 1:
        r = r + p
    p = p + p
    q = q // 2
```

a	$\times$	b	$=$	p	$\times$	q	$+$	r
7		6						
7	$\times$	6	$=$	7	$\times$	6	$+$	0
7	$\times$	6	$=$	14	$\times$	3	$+$	0
7	$\times$	6	$=$	28	$\times$	1	$+$	14
7	$\times$	6	$=$	56	$\times$	0	$+$	42

```
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r


    if q%2==1: r = r + p

    p = p + p

    q = q // 2

# r == a*b
```

calcul de la condition de vérification

```
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r


    if q%2==1: r = r + p

    p = p + p

    q = q // 2
    # a*b == p*q+r
# r == a*b
```

calcul de la condition de vérification

```
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r


    if q%2==1: r = r + p

    p = p + p
    # a*b == p * (q//2) + r
    q = q // 2
    # a*b == p*q+r
# r == a*b
```

calcul de la condition de vérification

```
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r


    if q%2==1: r = r + p
    # a*b == 2*p * (q//2) + r
    p = p + p
    # a*b == p * (q//2) + r
    q = q // 2
    # a*b == p*q+r
# r == a*b
```

calcul de la condition de vérification

```
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r
    # si q%2==1 alors a*b == 2*p * (q//2) + r + p
    #           sinon a*b == 2*p * (q//2) + r
    if q%2==1: r = r + p
    # a*b == 2*p * (q//2) + r
    p = p + p
    # a*b == p * (q//2) + r
    q = q // 2
    # a*b == p*q+r
# r == a*b
```

```
a*b == a*b + 0   et
pour tous p, q, r tels que a*b == p*q + r
    si q > 0 alors
        si q%2==1 alors a*b == 2*p * (q//2) + r + p
                  sinon a*b == 2*p * (q//2) + r
    sinon r == a * b
p,q,r = a,b,0
while  q > 0:  # invariant a*b==p*q+r
    # si q%2==1 alors a*b == 2*p * (q//2) + r + p
    #           sinon a*b == 2*p * (q//2) + r
    if q%2==1: r = r + p
    # a*b == 2*p * (q//2) + r
    p = p + p
    # a*b == p * (q//2) + r
    q = q // 2
    # a*b == p*q+r
# r == a*b
```

`a*b == a*b + 0` et
pour tous `p`, `q`, `r` tels que `a*b == p*q + r`
    si `q > 0` alors
        si q%2==1 alors `a*b == 2*p * (q//2) + r + p`
                    sinon `a*b == 2*p * (q//2) + r`
    sinon `r == a * b`

$$
\begin{array}{l}
\forall a, b. \\
\quad a \times b = a \times b + 0 \wedge \\
\quad \forall p, q, r.\ a \times b = p \times q + r \Rightarrow \\
\quad\quad (q > 0 \Rightarrow \\
\quad\quad\quad (q \equiv 1 \mod 2 \Rightarrow a \times b = a \times 2p * \lfloor q/2 \rfloor + r + p) \wedge \\
\quad\quad\quad (q \not\equiv 1 \mod 2 \Rightarrow a \times b = a \times 2p * \lfloor q/2 \rfloor + r)) \wedge \\
\quad\quad (q \leq 0 \Rightarrow \\
\quad\quad\quad r = a \times b)
\end{array}
$$

conclusion

plus de détails sur

http://why3.lri.fr/

- logiciel libre
- plus de 160 programmes prouvés
- documentation, notes de cours (y compris en français)

résumé

- 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