Introduction aux méthodes formelles

ANDSI, 2024-04-09

O. Hermant, Centre de recherche en informatique, Mines Paris, Université PSL

MINES PARIS | PSL

Exercice

Le programme suivant effectue-t-il le tri d'une liste ?

```
def tri(l):
    if l:
        return "list sorted"
    else:
        return 42
```

- oui
- non

Exercice

Le programme suivant effectue-t-il le tri d'une liste ?

```
def tri(l):
    if l:
        return "list sorted"
    else:
        return 42
```

- ▶ oui
- ▶ non

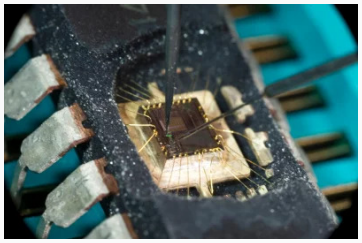
- Comment s'en convaincre, mieux qu'en agitant les mains ?
- C'est l'un des objectifs de cette soirée.

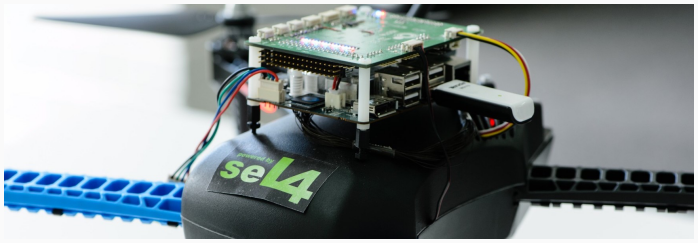
Les méthodes formelles : domaines d'application

- Sûreté

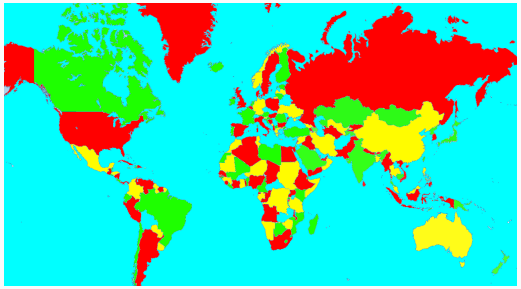


- Sécurité



- Mathématiques




Introduction

Ce que nous allons voir

- Votre intervenant
- pourquoi tout **doit-il** être si compliqué ?
- Le test
- L'approche logique
  - preuve et typage
  - spécification par assertions logiques
  - programmes corrects par construction
  - base de confiance
- La modélisation
  - model-checking
  - interprétation abstraite
  - langages synchrones
- La sécurité informatique
  - cryptographie
  - protocoles
- Conclusion

Votre intervenant

- Professeur au Centre de recherche en informatique, **Mines Paris, Université PSL**
- **Checheur associé** Inria Deducteam, LMF, U. Paris-Saclay
- conseil scientifique du GDR “Génie de la programmation et du logiciel” (GDR GPL, CNRS)
- Domaine de recherche : **preuve & méthodes formelles**
  - académique : thèses, publications, enseignement
  - recherche partenariale
- **Applications** : programmes corrects par construction (B) ; partage de preuves (Coq, Dedukti) ; formalisation de protocoles
- prénom.nom(a)minesparis.psl.eu

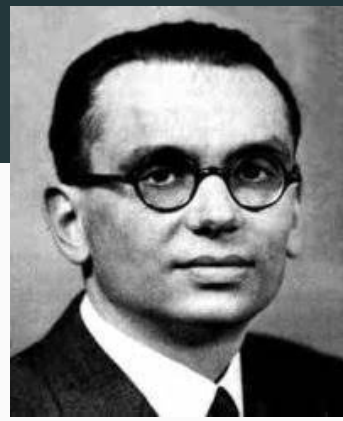
Méthodes formelles : que veut-on ?

- Cahier des charges :
  - Pour une propriété P et un programme p, a-t-on $P(p)$?
  - statiquement : on veut savoir si $P(p)$ avant de lancer p
  - automatiquement

Méthodes formelles : que veut-on ?

- Cahier des charges :
  - Pour une propriété P et un programme p, a-t-on $P(p)$?
  - statiquement : on veut savoir si $P(p)$ avant de lancer p
  - automatiquement
- hélas : **impossible**

Pourquoi tout doit-il être si compliqué ?



- Pour une propriété P et un programme p, a-t-on $P(p)$?
- Kurt Gödel et le théorème d'incomplétude : tout a des limites
- résultat clef du XXe siècle : pas si surprenant au XXIe
- conséquences en informatique :

Théorème (Rice, 1951)

Toute propriété sur les programmes qui est sémantique et non triviale est indécidable.

Théorème de Rice : le problème de l'arrêt

Indécidabilité du problème de l'arrêt

Il n'existe pas d'algorithme `seTermine(p,i)` qui *décide* si un programme `p` appliqué à un entrée `i` se termine.

Preuve par l'absurde :

- construire

```
def D(x):
    if seTermine(x,x):
        while(True):
            pass
    else:
        return 42
```

- se demander si `seTermine(D,D)`

Méthodes formelles : que peut-on ?

- Cahier des charges :
  - Un programme p a-t-il la propriété $P(p)$?
  - **statiquement** : savoir si $P(p)$ avant la mise en production
  - idéalement : **automatisé**
- Compromis à faire, selon le type de propriétés recherchées
  - exhaustivité : tests, model-checking
  - automatisation : assistants de preuve, logique de Hoare
  - complétude : démonstration automatique
  - précision : typage, model-checking, interprétation abstraite

Les tests

- excellente réponse à l'exercice initial !
- cela *fait* partie des méthodes formelles (e.g., U. Luxembourg)
- compromis exhaustivité
  - division flottante (Intel Pentium, 1994)
  - Coût officiel : $475,000,000
  - ou encore : AMD Phenom (2008), Intel Sandy Bridge (2011)
  - https://videotheque.cnes.fr/index.php?urlaction=doc&id_doc=16001&rang=1

Dijkstra, 1972

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

De la preuve

Assistants de preuve

- au lieu de tester $P(0), P(1) \ldots,$ **démontrer** $\forall n, P(n)$
- faire confiance aux ordinateurs
  - langage : spécifications, preuves
  - règles de raisonnement mécaniques (voire mécanisables)
  - plus stricts, fatiguent moins vite
- exemple typique :

T. Hales et al (Conjecture de Kepler)

The original proof, announced in 1998 and published in 2006, is long and complex. The process of [...] review did not end with the publication of the proof.

- extension de la notion de preuve : les ordinateurs, téléscopes des temps modernes
- **outils** : Coq, Isabelle, PVS, Mizar...

Démonstrations en pratique : l'outil Coq

- [elementary.v]
- démo live
- essayable à la maison
  - Option 1: installer Coq et Coqide.
  - Option 2: dans votre navigateur :
    https://coq.vercel.app/scratchpad.html

Coq : que retenir ?

- outil français (copie américaine : L∃∀N)
- énoncer, démontrer, manipuler les preuves
- **vérification** mécanisée
- **construction** automatisable à un certain point
- applications :
  - mathématiques — du lycée (LiberAbaci) aux Médailles Fields (V. Voevodsky)
  - preuve de programme
- degré de certitude extrêmement élevé mais jamais 100%

Le problème de la base de confiance

La base de confiance

- cadre logique puissant (trop ?) $\Rightarrow$ incohérences potentielles (et c'est "indémontrable" : cf. Gödel)
- implémentation (30,000 LOC)
- (compilateur du) langage OCaml
- système d'exploitation (e.g., intégrité)
- microprocesseur
- rayons cosmiques, particules α

- mitigation et minimisation possible
- élimination impossible

Typage fort

- impossible de prouver :

$$\forall n \quad , \sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

Typage fort

- possible de prouver :

$$\forall n : \text{nat} , \sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

- en programmation : **déclarer et vérifier les types**

```
Fixpoint sum (n : nat) : nat := match n with
  | 0     =>  0
  | (S n) =>  (S n) + (sum n)
  end.
```

Typage fort

- possible de prouver :

$$\forall n : \text{nat} , \sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

- en programmation : **déclarer et vérifier les types**

```
Fixpoint sum (n : nat) : nat := match n with
  | 0     =>  0
  | (S n) =>  (S n) + (sum n)
  end.
```

- Ce programme de tri de liste est-il *bien typé* ? [un_type.py]

```
def tri(l : list) -> list:
    if l:
        return "list sorted"
    else:
        return 42
```

  - Python : oui
  - typage statique (mypy): non

La correspondance preuves-programmes

- pont entre mathématiques constructives et programmes

Mathématiques	Informatique
Coq	OCaml
Propositions	Types, spécifications
Preuves	Programmes (λ-termes)

- synthèse de programmes corrects par construction

  $=$ recherche de preuve

  $=$ prouver une proposition/spécification P

- découverte majeure et graduelle : '60 $\sim$ '80

La puissance des types (ordre décroissant)

1. **assistants de preuve** : types très riches ($\equiv$ spécification précise) + preuve
2. **langages fonctionnels** (OCaml, Haskell) : types riches + inférence forte

R. Milner

"A well-typed program cannot go wrong."

3. **typage statique** (C++, Java, Rust...) : types + inférence
4. **typage dynamique** (Python, javascript, Perl, PHP) : ?

G. van Rossum

"I learned a painful lesson, that for small programs dynamic typing is great, for large programs you have to have a more disciplined approach and it helps if the language actually gives you that discipline, rather than telling you 'Well you can do whatever you want'."

Exemples d'utilisation des langages fonctionnels

- extraction Coq $\rightarrow$ OCaml : "oubli de la preuve, préservation du programme, l'efficacité en plus"
- programmes incroyablement concis : moins de lignes = plus de réflexion et moins de bugs
- modes de pensée algébriques (et non pas procéduraux)
- réalisations dans tous les domaines (e.g., Meta), en particulier
  - finance
  - CompCert : compilateur certifié... par rapport à quoi ? [C99-standard.pdf]
  - sel4 : micro-kernel certifié

Limites du typage

- une fonction (de tri ?) bien typée :

```
def tri(l : list) -> list :
    return l
```

- passe un grand nombre de tests

Limites du typage

- une fonction (de tri ?) bien typée :

```
def tri(l : list) -> list :
    return l
```

- passe un grand nombre de tests
- solutions :
  - plus de types (Coq, etc) :
    `list[A implements Comparable] -> sorted_list[A]`
  - utiliser la logique à la manière des assert de [un_test.py], amélioré :
  - pouvoir exprimer “assert sorted(l)” (expressivité logique)
  - pouvoir *démontrer* ces assertions $\neq$ vérifier à l'exécution
  - sans polluer le code

La Logique de Hoare

Triplets de Hoare

$$\{P\}\,\texttt{instr}\,\{Q\}$$

Si la propriété P (précondition) est vraie, alors la propriété Q (postcondition) sera vraie après exécution de l'instruction `instr`, si celle-ci termine.

- C.A.R. Hoare (1969) et W. Floyd (1967)
- règles de déduction pour la validité des triplets

Exemples/Exercices

- souvent on sait seulement ce que l'on veut

  $\{ ?? \}$

  `x = y;`

  $\{ ?? \}$

  `x = x + 1;`

  $\{ x > 0 \}$

Exemples/Exercices

- souvent on sait seulement ce que l'on veut

  $\{ ?? \}$

  `x = y;`

  $\{ x + 1 > 0 \}$

  `x = x + 1;`

  $\{ x > 0 \}$

Exemples/Exercices

- souvent on sait seulement ce que l'on veut

  $\{y + 1 > 0\}$

  `x = y;`

  $\{ x + 1 > 0 \}$

  `x = x + 1;`

  $\{ x > 0 \}$
- $\{$ `y > 42` $\}$ serait suffisant, mais on cherche

Plus faible précondition

wp(`instr`, Q) est la propriété telle que :

- $\{\text{wp}(\texttt{instr}, Q)\}\ \texttt{instr}\ \{Q\}$ est vrai
- si $\{P\}\ \texttt{instr}\ \{Q\}$ est vrai, alors $P \Rightarrow \text{wp}(\texttt{instr}, Q)$

- calcul de plus faible précondition non parfaitement automatisable (invariants de boucle)

En pratique : un tri prouvé correct

- l'outil Frama-C (démo)

En pratique : un tri prouvé correct

- l'outil Frama-C (démo)
- preuve automatique (solveurs SMT), assistée (Coq) et Why3
- note : spécification pas encore 100% complète

```
1          void sort(int *t, int n) {
             for(int i=0;i<n;i++){
3              t[i] = i;
             }
5          }
```

- demanderait : entrée et sortie sont des permutations, etc...

En pratique : un tri prouvé correct

- l'outil Frama-C (démo)
- preuve automatique (solveurs SMT), assistée (Coq) et Why3
- note : spécification pas encore 100% complète

```
1          void sort(int *t, int n) {
             for(int i=0;i<n;i++){
3              t[i] = i;
             }
5          }
```

- demanderait : entrée et sortie sont des permutations, etc...
- mais on peut débusquer certains bugs vicieux

```
1          void quickSort(int[] arr, int low, int high){
             if (low < high) {
3              pi = partition(arr, low, high);
               quickSort(arr, low, pi);
5              quickSort(arr, pi, high);
             }
7          }
```

Des programmes corrects par construction

Quelques chiffres

- coût des bugs: \$64 milliards par an (USA)
  - pédale d'accélérateur Toyota Camry : **\$1.2 milliard**
- coût de la vérification
  - \$10 milliards par an pour les tests (USA)
  - débugger 50 kLOC : 60 jours, \$30,000 + coûts indirects
  - $>$ 50% du coût d'un système critique
  - industrie du software : $\sim$ 30%
- coût d'un bug
  - 1 à la conception
  - 5-15 pendant les tests unitaires
  - 15-90 à l'intégration
  - 50-200 en production
- répartition des bugs (secteur automobile, Bosch)
  - 60% spécification, 20% design, 10% code, 10% système

Certifications Critères Communs

- délivrées par l'ANSSI
- rapports établis par des labos agrées (CESTI)
- niveau de sûreté le plus élevé
  - demande spécification formelle du code
  - sur certains outils qualifiés
- différents niveau de sûreté :
  - Evaluation Assurance Levels (EAL 1–7)
- réglementaire dans certains domaines critiques
  - transport ferroviaire et aérien
  - énergie nucléaire
  - carte à puce (non réglementaire)
  - santé ? voiture ?

Des programmes corrects par construction

Principe:

- écrire la spécification
- générer le “code”
  - automatiquement
  - ou par raffinements successifs
- **forcer l’adéquation** specs-code
  - avec des étapes elles-mêmes *prouvées correctes*
  - avec des preuves
  - dans un formalisme mathématique / ordinateur

La méthode B (∼1990)

- de la spécification à l'implémentation dans un cadre logique
- industrie ferroviaire (Siemens, Mitsubishi, RATP)
- http://www.atelierb.eu/

Trains automatiques en réseau ouvert

- 1998 : ligne 14 (Paris)
- 2005 : ligne 1 (Paris)
- 2006 : ligne L (New-York)
- 2009 : ligne 9 (Barcelone)

La méthode B

- de la spécification à l'implementation
- logique du premier ordre + théorie des ensembles typée
- étapes successives de raffinements
- correction de chaque étape
- programmes **corrects par construction**
- milliers d'obligations de preuve
- manuelles ou déchargées à des prouveurs automatique

Approche sceptique

- fiabilité des prouveurs automatiques ?
- **demander des certificats**
- revérification par une **tierce partie**
- BCoq, Dedukti
- assistants de preuves petits et contrôlés
- pas exempts de bugs non plus (base de confiance)

TLA+

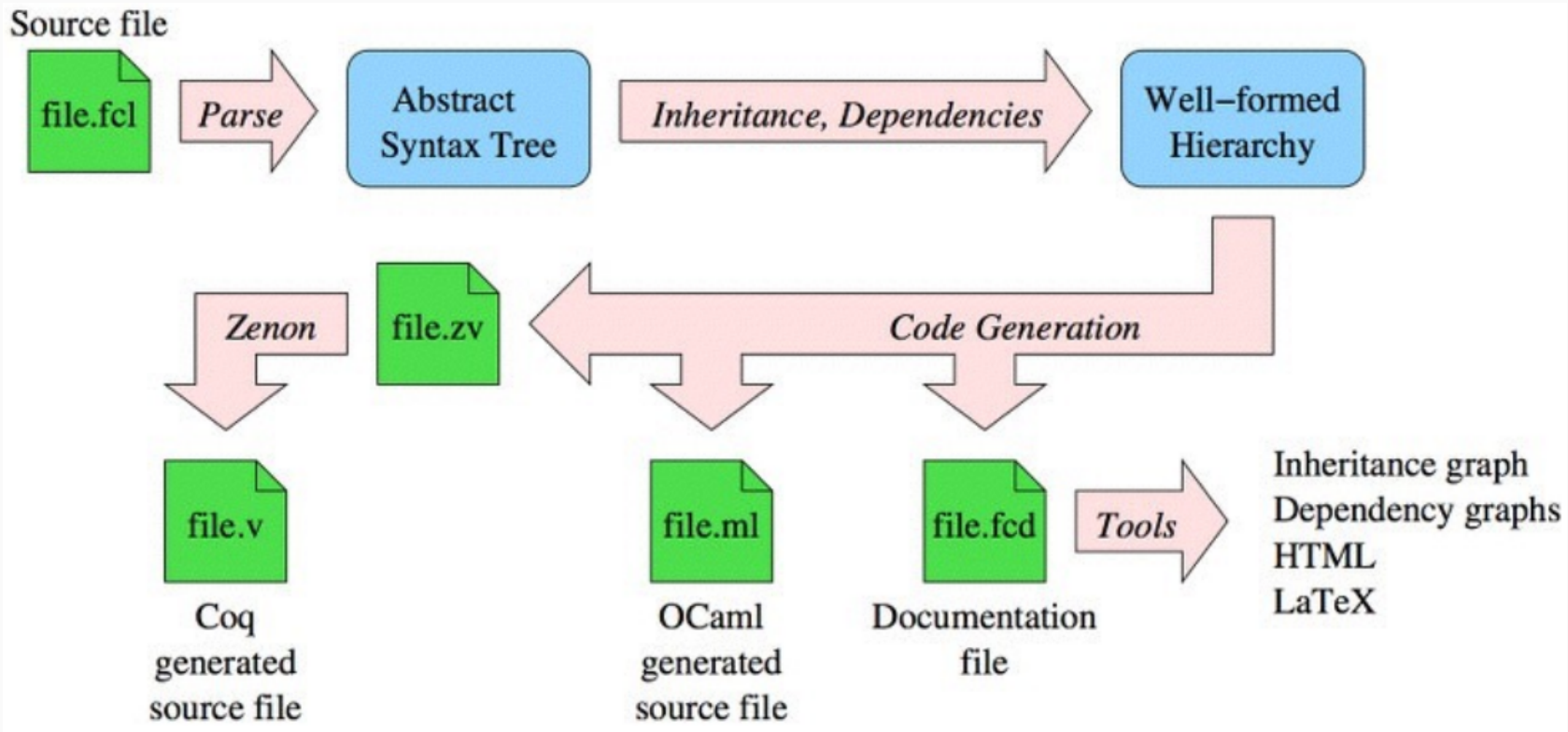
- Temporal Logic of Actions
- L. Lamport (Prix Turing), S. Merz
- base théorie des ensembles
- propriétés temporelles exprimables (vivacité, famine)
- démonstration automatique pour décharger les preuves

L'atelier FoCaLiZe

- LIP6 (U. P. & M. Curie), Cedric (CNAM), SAMOVAR (ENSIIE), ENSTA, Inria (90/00)
- **traits objets** (héritage)
- spécification et implémentation
  - preuves déléguées à Zenon
- compilation
  - vers OCaml (programme)
  - vers Coq (preuve)
  - adéquation Coq–OCaml
- réalisations
  - SafeRiver
  - Bertin Technologies
  - politique de sécurité aéroports
  - FoCaLiZe $\rightarrow$ UML

L'atelier FoCaLiZe

- http://focalize.inria.fr/



Confiance dans la base de confiance

Quel **apport des méthodes formelles** pour :

1. compilation,
2. code compilé,
3. système d'exploitation,
4. matériel?

Compilation certifiée

- code source: C, et code exécuté: assembleur. Adéquation ?
- selon quels critères ?
  - source (standard C90) vs. assembleur (sémantique)

$$\text{CompCert} : \text{code C} \rightarrow \text{assembleur}$$

$$\textbf{Théorème:} \quad \forall\ p.\ \text{CompCert}(p) \text{ simule } p$$

- http://compcert.inria.fr/
- quelques limitations, moins d'optimisations, mais :
  - aucune erreur trouvée [PLDI2011]
  - tests extensifs, aux limites de la norme
  - autres compilateurs tous erronés
  - **performances ok**

Conception matérielle

- spécification et implantation: adéquation ?
- matériel : circuit imprimé, portes logiques, processeur
  - **coût de la compilation**
  - coût bug
- vérification formelle le plus en amont possible
- deux exemples :
  - http://www.cl.cam.ac.uk/ jrh13/hol-light/ (J. Harrisson, Intel)
  - Esterel ('80 $\sim$ '90, G. Berry)

Langages synchrones

- **hypothèse synchrone** : le système réagit instantanément aux événements
- systèmes embarqués temps-réel
- garanties sur le pire temps d'exécution (WCET)
- outil SCADE
  - développement certifié : Airbus, Dassault...
- **Lustre**
  - langage de base de SCADE
  - compilateurs certifiés en Coq

De la modélisation

Systèmes réactifs

Programmes classiques

- calculent
- terminent
- renvoient un résultat
- données complexes, exécution séquentielle

Programmes réactifs

- *doivent* ne pas terminer
- ne retournent rien
- données simples, exécution distribuée
- inclure le modèle physique dans la conception ?

Exemples

Systèmes d'exploitation, systèmes contrôle-commande, $\cdots$

Vérifier un système réactif

Programmes classiques

- prédicats logiques complexes, pas d'aspect temporel
- "le tableau est trié"

Programmes réactifs

Prédicats plus simples, aspect temporel primordial :

- **vivacité** : "*si un processus fait une demande, l'OS finira par la traiter*"
- **équité** : "*si un processus fait des demandes en nombre arbitraire, l'OS finira par les traiter toutes*"
- **accessibilité** : "*il est toujours possible de revenir à l'état initial*"

Model Checking

Model Checking

Technique de vérification automatisée pour les systèmes réactifs

Principes

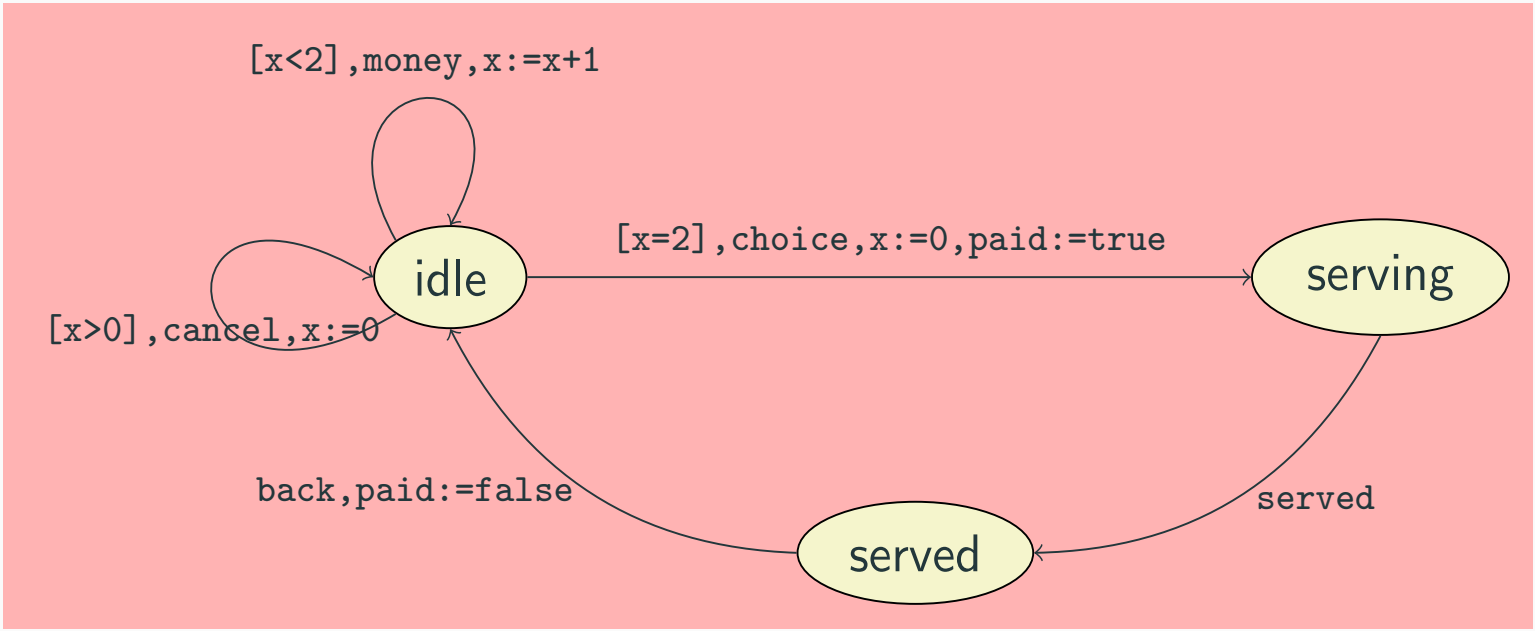
1. construire un modèle M du système (automatisé ou non)
2. exprimer la propriété φ
3. a-t-on $M \models \varphi$?
   - oui, ok. Attention cependant ! M n'est qu'un modèle.
   - non, il y a un contre-exemple. À rejouer sur le système réel
     - vrai bug
     - sinon raffiner M

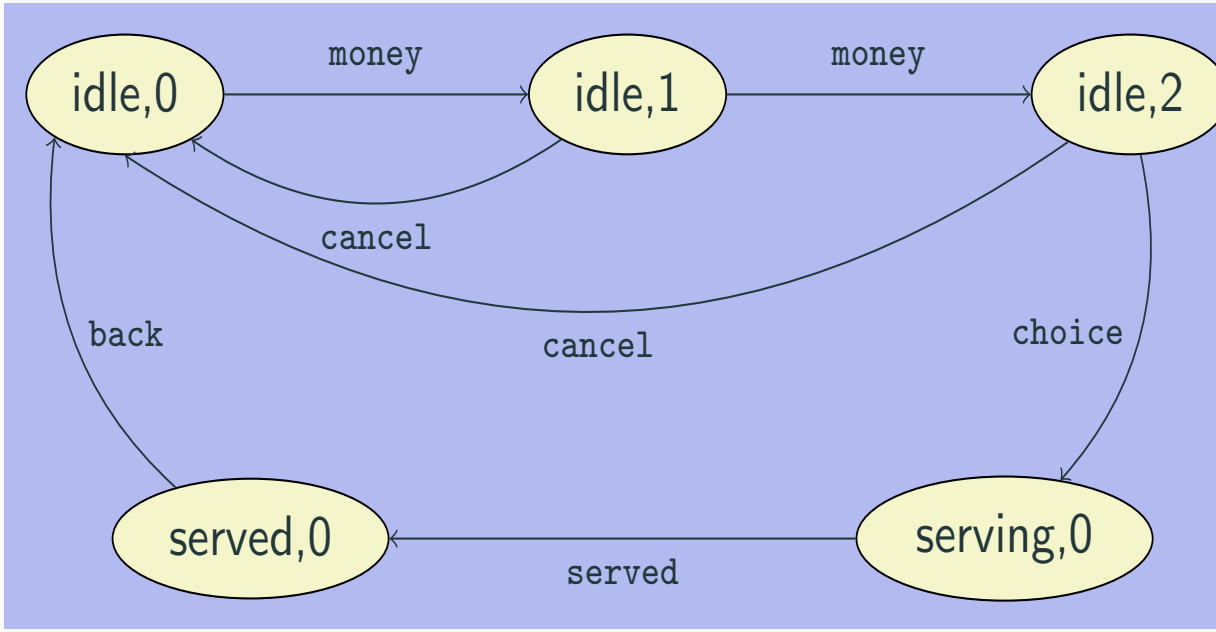
- applicable au moment du design et de la validation
- automatisé
- moindre coût que d'autres méthodes
- traquage de bugs plus exhaustif que le test

Historique

- 1975 : vérification traditionnelle mal adaptée aux systèmes réactifs
- 1977 : Pnueli propose d'utiliser la logique temporelle
- 1981 : CTL Model Checking par Clarke, Sifakis, Emerson
- '80 & '90 : théorie
- '90 & '00 : amélioration des performances et extensions (probabiliste, temps)
- '00 : standardisation, adoption industrielle
  - design de processeurs (Intel, ...)
  - language PSL standardisé
  - *software* Model Checking (MS)
- '07 : Prix Turing (Clarke, Sifakis, Emerson)

Machines à états $\rightsquigarrow$ Systèmes de transitions d'états



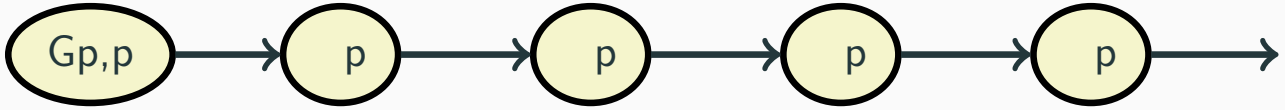


Vérifier un système de transitions

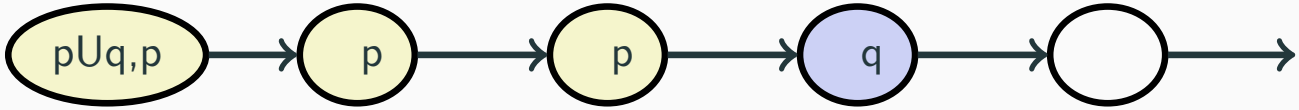
- exprimer une succession d'événements le long d'un chemin d'exécution
  - X – neXt: "à la prochaine étape"

  - F – Finally: "à un moment dans le futur"

  - G – Globally: "à tout moment dans le futur"



  - U – Until: "jusqu'à ce que"



- quantification sur les chemins futurs (tous, au moins un)
- défi du Model-Checking : **explosion des états**
  - exemple : 10 variables sur 8 bits: 10^{256} possibilités

Exemples

- **Accessibilité**.
  - On peut atteindre une certaine situation
  - $\mathsf{F}(x = 0)$
- **Invariance**.
  - Tout état respecte une certaine propriété
  - $\mathsf{G}\ \neg(x = 0)$
- **Vivacité**.
  - Une bonne chose finit toujours par arriver
  - $\mathsf{G}(p \Rightarrow \mathsf{F}q)$

Parallélisme et concurrence

- définir un **modèle d'exécution et mémoire** (e.g., π-calcul)
- calculs de dépendances (instructions et données)
- **propriétés difficiles** à assurer :
  - au niveau du code et, pire, de sa preuve
  - race condition, deadlocks, fairness, vivacité...
- transformations de code qui respectent la sémantique (e.g., PIPS)
- analyses de traces d'exécution (forensics)

Interprétation abstraite

- objectif : **analyse des valeurs** dans un programme
- approximations nécessaires

$$\begin{array}{rcl} \text{domaine concret} & \mapsto & \text{domaine abstrait} \\ \text{nombres flottants} & \rightsquigarrow & \text{intervalles} \end{array}$$

- **pertes** : précision, pouvoir expressif **& profits** : automatisé
- exemple d'analyse

`if (test)`
`  x = 1`
`else`
`  x = 2`
`y = 2/x`
	$x \in [1, 1]$

Interprétation abstraite

- objectif : **analyse des valeurs** dans un programme
- approximations nécessaires

$$\begin{array}{rcl} \text{domaine concret} & \mapsto & \text{domaine abstrait} \\ \text{nombres flottants} & \rightsquigarrow & \text{intervalles} \end{array}$$

- **pertes** : précision, pouvoir expressif **& profits** : automatisé
- exemple d'analyse

`if (test)`	
`  x = 1`	$x \in [1, 1]$
`else`	
`  x = 2`	$x \in [2, 2]$
`y = 2/x`	

Interprétation abstraite

- objectif : **analyse des valeurs** dans un programme
- approximations nécessaires

$$\begin{array}{rcl} \text{domaine concret} & \mapsto & \text{domaine abstrait} \\ \text{nombres flottants} & \rightsquigarrow & \text{intervalles} \end{array}$$

- **pertes** : précision, pouvoir expressif **& profits** : automatisé
- exemple d'analyse

`if (test)`	
`  x = 1`	$x \in [1,1]$
`else`	
`  x = 2`	$x \in [2,2]$
`y = 2/x`	$x \in$?

  - besoin de l'union convexe : $[1,1] \uplus [2,2] = [1,2]$

Interprétation abstraite

- objectif : **analyse des valeurs** dans un programme
- approximations nécessaires

$$\begin{array}{rcl} \text{domaine concret} & \mapsto & \text{domaine abstrait} \\ \text{nombres flottants} & \rightsquigarrow & \text{intervalles} \end{array}$$

- **pertes** : précision, pouvoir expressif **& profits** : automatisé
- exemple d'analyse

`if (test)`	
`  x = 1`	$x \in [1,1]$
`else`	
`  x = 2`	$x \in [2,2]$
`y = 2/x`	$x \in [1,2]$

  - besoin de l'union convexe : $[1,1] \uplus [2,2] = [1,2]$
  - **absence d'erreur à l'exécution** : pas de division par 0
  - pas à tous les coups

Outils d'interprétation abstraite

- VERASCO : analyse statique pour le code produit par CompCert
  - absence d'erreur à l'exécution
  - prouvé en Coq
- ASTRÉE
- https://www.absint.com/ : commercialisation de certains outils (ASTRÉE, CompCert...)
- PIPS : framework de compilation source-à-source (C, Fortran)
  - parallélisation, optimisations automatiques
  - CRI, Mines Paris (1990–2010)
  - https://pips4u.org/

Questions de représentation : les nombres flottants

- combien vaut `0.1` ? [`valeur_un.py`]

N	droite : $\frac{1}{1^2} + (\frac{1}{2^2} + (\frac{1}{3^2} + \cdots))$	gauche : $((\frac{1}{1^2} + \frac{1}{2^2}) + \frac{1}{3^2}) + \cdots$
2^{10}	1.643957027355847699 82	1.643957027355849032 09
2^{20}	1.644931131731734552 5246	1.644931131731734961 0867
2^{30}	1.644934065916903831 45	1.644934057834575025 23
4.2^{30}	1.644934066615395762 41	1.644934057834575025 23
32.2^{30}	1.644934066790018745 15	1.644934057834575025 23
limite ($\frac{\pi^2}{6}$)	1.6449340668482264 3	1.6449340668482264 3

Questions de représentation : les nombres flottants

- combien vaut `0.1` ? [valeur_un.py]
- combien vaut `1 + 1e-30` ? [epsilon.py]

N	droite : $\frac{1}{1^2} + (\frac{1}{2^2} + (\frac{1}{3^2} + \cdots))$	gauche : $((\frac{1}{1^2} + \frac{1}{2^2}) + \frac{1}{3^2}) + \cdots$
2^{10}	1.643957027355847699 82	1.643957027355849032 09
2^{20}	1.644931131173455252 46	1.644931131173496108 67
2^{30}	1.644934065916903831 45	1.644934057834575025 23
4.2^{30}	1.644934066615395762 41	1.644934057834575025 23
32.2^{30}	1.644934066790018745 15	1.644934057834575025 23
limite ($\frac{\pi^2}{6}$)	1.644934066848226 43	1.644934066848226 43

Questions de représentation : les nombres flottants

- combien vaut `0.1` ? [valeur_un.py]
- combien vaut `1 + 1e-30` ? [epsilon.py]
- quelle est la meilleure manière de calculer ?

$$\sum_{k=1}^{N} \frac{1}{k^2}$$

- (converge vers $\pi^2/6$) [somme_entiers.py]

N	droite : $\frac{1}{1^2} + (\frac{1}{2^2} + (\frac{1}{3^2} + \cdots))$	gauche : $((\frac{1}{1^2} + \frac{1}{2^2}) + \frac{1}{3^2}) + \cdots$
2^{10}	1.643957027355847699 82	1.643957027355849032 09
2^{20}	1.644931131173455252 46	1.644931131173496108 67
2^{30}	1.644934065916903831 45	1.644934057834575025 23
4.2^{30}	1.644934066615395762 41	1.644934057834575025 23
32.2^{30}	1.644934066790018745 15	1.644934057834575025 23
limite ($\frac{\pi^2}{6}$)	1.6449340668482264 3	1.6449340668482264 3

Questions de représentation : les nombres flottants

- combien vaut `0.1` ? [valeur_un.py]
- combien vaut `1 + 1e-30` ? [epsilon.py]
- quelle est la meilleure manière de calculer ?

$$\sum_{k=1}^{N} \frac{1}{k^2}$$

  - (converge vers $\pi^2/6$) [somme_entiers.py]
- l'addition n'est pas associative !

N	droite : $\frac{1}{1^2} + (\frac{1}{2^2} + (\frac{1}{3^2} + \cdots))$	gauche : $((\frac{1}{1^2} + \frac{1}{2^2}) + \frac{1}{3^2}) + \cdots$
2^{10}	1.643957027355847699 82	1.643957027355849032 09
2^{20}	1.644931131173455252 46	1.644931131173496108 67
2^{30}	1.644934065916903831 45	1.644934057834575025 23
4.2^{30}	1.644934066615395762 41	1.644934057834575025 23
32.2^{30}	1.644934066790018745 15	1.644934057834575025 23
limite ($\frac{\pi^2}{6}$)	1.644934066848226 43	1.644934066848226 43

Encore des histoires de bug

- 1991, Guerre du Golfe

Encore des histoires de bug

- 1991, Guerre du Golfe

- 2007, F22

Méthodes formelles et sécurité

De la sûreté à la sécurité

- **sûreté de fonctionnement**
  - un programme respecte ses spécifications
  - étudié et compris
  - particulièrement les *propriétés fonctionnelles*
  - mais aussi : concurrence, circuits électroniques, réactifs, synchrone
- **robustesse**
  - nécessite des modèles
- **sécurité informatique**
  - quels modèles ?

Sécurité des SI

- **Confidentialité** : information disponible uniquement pour les entités autorisées
- **Intégrité** : information non corrompue, activement ou passivement
- **Authenticité** : origine de l'information vérifiable

Mais aussi :

- politiques de sécurité
- cybersécurité physique
- protocoles
- données personnelles (RGPD)

Cryptographie

- modélisation des types d'attaques (Dolev-Yao)
- preuves de sécurité : EasyCrypt
  - logique de Hoare probabiliste
  - modèle de calcul réaliste ($\approx$ celui des cryptographes)
  - preuves formelles de sécurité
  - en présence de code adverse (modèle "attaquant qui transporte le message")
- approche ressemblant à Frama-C, avec Why3 et décharge de preuve

Confidentialité

Modélisation, formalisation et vérification de :

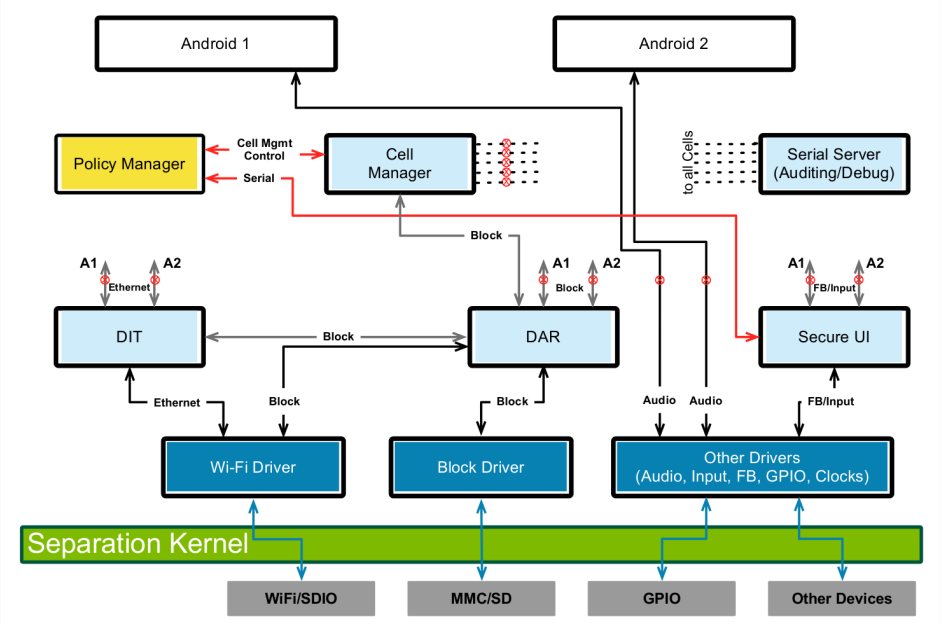
- differential privacy : démontrer l'impact 0 de l'aspect personnel des données
- traçabilité des données
- droits d'accès et contrôle des privilèges

Modélisation de protocoles

- communication
- sécurité
  - passeport biométrique
  - vote électronique
  - cartes à puce THALES
- aspects multiples : cryptographiques, confidentialité, échange des données, non interférence, authenticité...
- aspects semi-formels :
  - politiques de sécurité (aéroports)
  - textes de loi (impôts, consolidation du droit...)
- à chaque fois on trouve des failles !

Système d'exploitation

- sel4 microkernel/OKL4 microviseur, https://sel4.systems/
- noyau de séparation OS / matériel

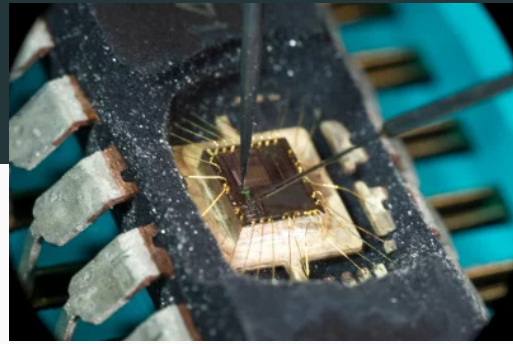


- sécurité et correction prouvées avec des assistants de preuve
- meilleures performances, coûts moindres (> EAL7)

Question coûts

$362 per LOC for the correctness + $78 per line for the security proofs.

Vers le bas niveau



- **analyse de code binaire** (interprétation abstraite, logique déductive, exécution symbolique)
  - BinSec
    - escalade de privilèges
    - détection de vulnérabilités
  - Laboratoire Haute Sécurité (Loria) : analyses de flots de données et de comportements
- **cybersécurité physique**
  - modèles d'attaque et de pouvoir de l'adversaire
  - modélisation du code ($\sim$ système de transitions) et mutations
  - robustesse par rapport à bit flip

Conclusion

Conclusion

- “démontrer un fonctionnement correct”
  - chaque mot a son importance !
- panorama très large :
  - type de propriété $\leftsquigarrow$ type de méthode
  - base commune : mathématiques et logique
- 21e siècle : ne plus laisser l'écriture de programmes aux humains
  - les specs, oui ! (là aussi, attention aux bugs)
- vraie expertise européenne
- coûts : encore élevés, mais en baisse

Conclusion

- le bug **informatique** n'existe pas

Problem Exists Between Chair and Keyboard

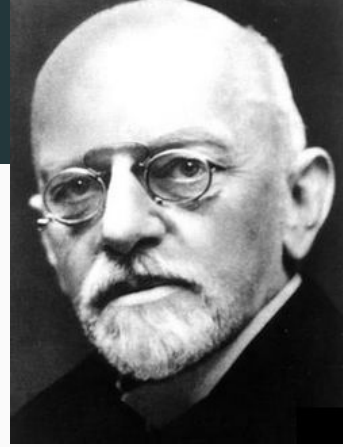
Entre la chaise et le clavier : erreurs humaines, que l'ordinateur amplifie.

- contrer ceci : formaliser
- le mot de la fin

D. Knuth

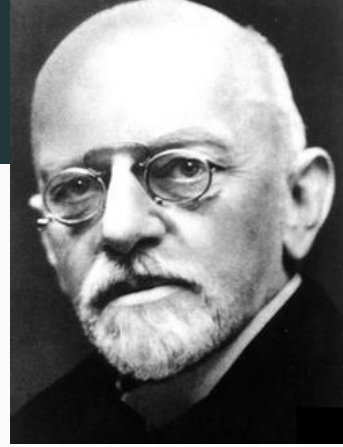
Beware of bugs in the above code; I have only proved it correct, not tried it.

Théorème d'incomplétude de Gödel : l'arithmétique



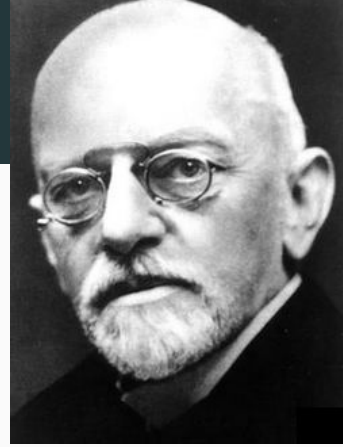
- qu'est-ce qu'un nombre entier ?

Théorème d'incomplétude de Gödel : l'arithmétique



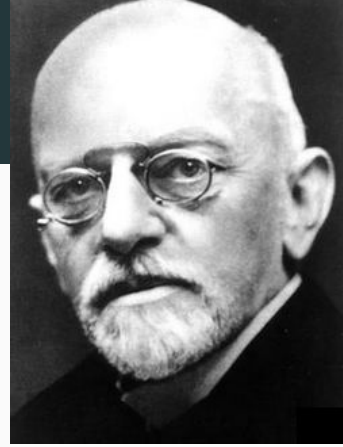
- qu'est-ce qu'un nombre entier ?
  - 0, 1...
  - généralisation : 0 est entier, si n est entier $n+1$ est entier
- pourquoi $\mathbb{R}$ n'est-il pas l'ensemble des nombres entiers ?

Théorème d'incomplétude de Gödel : l'arithmétique



- qu'est-ce qu'un nombre entier ?
  - 0, 1...
  - généralisation : 0 est entier, si n est entier $n+1$ est entier
- pourquoi $\mathbb{R}$ n'est-il pas l'ensemble des nombres entiers ?
  - $\sqrt{2}$, π, e : irrationels
  - $\mathbb{R}$ a la chaîne infinie descendante $0, -1, -2, \cdots$
  - $\mathbb{R}$ est trop gros : on veut le **plus petit ensemble** contenant 0 et clos par successeur ($n \Rightarrow n+1$)
  - être entier $\equiv$ respecter le **principe de récurrence**

Théorème d'incomplétude de Gödel : l'arithmétique



- qu'est-ce qu'un nombre entier ?
  - 0, 1...
  - généralisation : 0 est entier, si n est entier $n+1$ est entier
- pourquoi $\mathbb{R}$ n'est-il pas l'ensemble des nombres entiers ?
  - $\sqrt{2}$, π, e : irrationels
  - $\mathbb{R}$ a la chaîne infinie descendante $0, -1, -2, \cdots$
  - $\mathbb{R}$ est trop gros : on veut le **plus petit ensemble** contenant 0 et clos par successeur ($n \Rightarrow n+1$)
  - être entier $\equiv$ respecter le **principe de récurrence**
- axiomatisation de l'arithmétique par Peano (19e siècle)
- 2$^{\text{e}}$ sur les **23 problèmes de Hilbert** (1900) : cohérence de l'arithmétique ? (= non contradictoire)
- Gentzen (1936) : l'arithmétique est cohérente
- Gödel (1931) : la cohérence est impossible à démontrer

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres (+, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres ($+$, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)
- construire deux fonctions (partielles), inverses :

$$\lceil . \rceil \ : \ \texttt{Syntaxe} \ \to \ \mathbb{N} \qquad \lfloor . \rfloor \ : \ \mathbb{N} \ \to \ \texttt{Syntaxe}$$

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres ($+$, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)
- construire deux fonctions (partielles), inverses :

$$\lceil . \rceil \ : \ \texttt{Syntaxe} \ \to \ \mathbb{N} \qquad \lfloor . \rfloor \ : \ \mathbb{N} \ \to \ \texttt{Syntaxe}$$

- construire $\texttt{Dem}(n)$: *vraie* ssi "la proposition $\lfloor n \rfloor$ est prouvable"
- construire la fonction *app* : $app(n, m)$ est l'entier $\lceil (\lfloor n \rfloor (m)) \rceil$

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres ($+$, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)
- construire deux fonctions (partielles), inverses :

$$\ulcorner . \urcorner \ : \ \texttt{Syntaxe} \ \to \ \mathbb{N} \qquad \llcorner . \lrcorner \ : \ \mathbb{N} \ \to \ \texttt{Syntaxe}$$

- construire $\texttt{Dem}(n)$: *vraie* ssi "la proposition $\llcorner n \lrcorner$ est prouvable"
- construire la fonction *app* : $app(n, m)$ est l'entier $\ulcorner (\llcorner n \lrcorner (m)) \urcorner$
- la proposition qui fâche : $\Delta(n) = \neg \texttt{Dem}(app(n, n))$. "la formule $\llcorner n \lrcorner (n)$ n'est pas prouvable"

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres (+, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)
- construire deux fonctions (partielles), inverses :

$$\lceil . \rceil \ : \ \texttt{Syntaxe} \ \rightarrow \ \mathbb{N} \qquad \lfloor . \rfloor \ : \ \mathbb{N} \ \rightarrow \ \texttt{Syntaxe}$$

- construire $\texttt{Dem}(n)$: *vraie* ssi "la proposition $\lfloor n \rfloor$ est prouvable"
- construire la fonction *app* : $app(n, m)$ est l'entier $\lceil (\lfloor n \rfloor (m)) \rceil$
- la proposition qui fâche : $\Delta(n) = \neg\texttt{Dem}(app(n, n))$. "la formule $\lfloor n \rfloor (n)$ n'est pas prouvable"
- la question qui fâche, $G ::= \Delta(\lceil \Delta \rceil)$
  - $G \equiv \neg\texttt{Dem}(\lceil (\lfloor \lceil \Delta \rceil \rfloor (\lceil \Delta \rceil)) \rceil) \equiv \neg\texttt{Dem}(\lceil \Delta(\lceil \Delta \rceil) \rceil) \equiv \neg\texttt{Dem}(\lceil G \rceil)$
  - si G est prouvable, G est vraie (thm de correction), donc (par définition de $\texttt{Dem}$), $\lfloor \lceil G \rceil \rfloor = G$ n'est pas prouvable. $\perp$

Théorème d'incomplétude de Gödel

- travail d'encodage sur des nombres ($+$, *,récurrence)
  - est-ce si étonnant, au 21e siècle ? (penser à Coq)
- construire deux fonctions (partielles), inverses :

$$\lceil . \rceil \ : \ \texttt{Syntaxe} \ \rightarrow \ \mathbb{N} \qquad \lfloor . \rfloor \ : \ \mathbb{N} \ \rightarrow \ \texttt{Syntaxe}$$

- construire $\texttt{Dem}(n)$: *vraie* ssi "la proposition $\lfloor n \rfloor$ est prouvable"
- construire la fonction *app* : $app(n, m)$ est l'entier $\lceil (\lfloor n \rfloor (m)) \rceil$
- la proposition qui fâche : $\Delta(n) = \neg\texttt{Dem}(app(n, n))$. "la formule $\lfloor n \rfloor (n)$ n'est pas prouvable"
- la question qui fâche, $G ::= \Delta(\lceil \Delta \rceil)$
  - $G \equiv \neg\texttt{Dem}(\lceil (\lfloor \lceil \Delta \rceil \rfloor (\lceil \Delta \rceil)) \rceil) \equiv \neg\texttt{Dem}(\lceil \Delta(\lceil \Delta \rceil) \rceil) \equiv \neg\texttt{Dem}(\lceil G \rceil)$
  - si G est prouvable, G est vraie (thm de correction), donc (par définition de $\texttt{Dem}$), $\lfloor \lceil G \rceil \rfloor = G$ n'est pas prouvable. $\bot$
- G est donc indémontrable, càd $\neg\texttt{Dem}(\lceil G \rceil)$ est vraie (i.e., G).
- G est vraie (dans $\mathbb{N}$) et non prouvable : $\texttt{Peano}$ **incomplet**.

Second théorème d'incomplétude

- mal de crâne ? Gödel et Hilbert ont pensé à vous :

Second théorème d'incomplétude

La cohérence de l'arithmétique est indémontrable (en arithmétique seule).

Second théorème d'incomplétude

- mal de crâne ? Gödel et Hilbert ont pensé à vous :

Second théorème d'incomplétude

La cohérence de l'arithmétique est indémontrable (en arithmétique seule).

- $\texttt{Peano} \Rightarrow \mathit{coh}(\lceil \texttt{Peano} \rceil)$ non prouvable !
  - tout à fait démontrable dans un système *plus puissant* que l'arithmétique de Peano (Gentzen)
- arithmétique (probablement) cohérente : $\mathit{coh}(\lceil \texttt{Peano} \rceil)$ est donc vraie.

Second théorème d'incomplétude

- mal de crâne ? Gödel et Hilbert ont pensé à vous :

Second théorème d'incomplétude

La cohérence de l'arithmétique est indémontrable (en arithmétique seule).

- $\texttt{Peano} \Rightarrow \mathit{coh}(\lceil \texttt{Peano} \rceil)$ non prouvable !
  - tout à fait démontrable dans un système *plus puissant* que l'arithmétique de Peano (Gentzen)
- arithmétique (probablement) cohérente : $\mathit{coh}(\lceil \texttt{Peano} \rceil)$ est donc vraie.
- **conséquences** :
  - les nombres entiers ne sont pas ce que l'on croit:
    - $\mathbb{N}$ n'est pas le seul modèle de l'arithmétique !
    - il y a d'autres *modèles*, avec des entiers non standards, où $\mathit{coh}(\lceil \mathcal{A} \rceil)$ et G sont fausses

Sources : images, citations, liens I (avril 2024)

#3

A380 (Wikipedia)	https://en.m.wikipedia.org/wiki/File:Airbus_A380_inbound_ILA_2006.jpg
Ligne 14 (Wikipedia)	https://fr.wikipedia.org/wiki/Ligne_14_du_m%C3%A9tro_de_Paris
Wall Street (pxhere)	https://pxhere.com/en/photo/910649
Centrale de Nogent (Wikipedia)	https://br.m.wikipedia.org/wiki/Restr:Nogent-sur-Seine-FR-10-centrale_nucl%C3%A9aire_le_soir-2.jpg
Puce décapée, sonde RF (T. Univ. München)	https://www.ce.cit.tum.de/en/eisec/research/invasive-attacks/
Passeport biométrique (Wikipedia)	https://fr.wikipedia.org/wiki/Passeport_fran%C3%A7ais
Microkernel Sel4 (Sel4)	https://sel4.systems/
Théorème des 4 couleurs (Wikipedia)	https://fr.wikipedia.org/wiki/Fichier:World_map_colored_using_the_four_color_theorem_including_oceans.png
Conjecture de Kepler (pxhere)	https://pxhere.com/fr/photo/642416

#5

GDR GPL	https://gdr-gpl.cnrs.fr/
conférence AFADL	https://gdrgpl2024.sciencesconf.org/resource/page/id/4
Mines Paris	
Inria Deducteam	https://deducteam.gitlabpages.inria.fr/
Dedukti	https://deducteam.github.io/

#7

Kurt Gödel (Univ. St Andrews)	https://mathshistory.st-andrews.ac.uk/Biographies/Godel/

Sources : images, citations, liens II (avril 2024)

#10

U. du Luxembourg — https://www.uni.lu/snt-fr/

Bug Pentium 1994 — https://aconit.inria.fr/omeka/exhibits/show/validite/bug/bug-pentium.html

Bugs Phenom Sandy Bridge — https://www.zeden.net/actu/12688-AMD-Phenom-bug-patch-et-baisse-de-perfs
https://www.silicon.fr/bug-du-sandy-bridge-les-constructeurs-rappellent-les-pc-affectes-44429.html

Dijkstra, *The Humble Programmer*, 1972 — CACM 15 (10), pp. 859–866, https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

#11

Empilement de fruits (pickpik) (pickpic) — https://www.pickpik.com/fruit-fruit-stand-fruits-market-stall-healthy-food-9887

T. Hales et al., *A revision of the proof of the Kepler conjecture*, 2009 — Discrete Comput Geom 44, pp. 1–34 (2010). https://doi.org/10.1007/s00454-009-9148-4

#12

The Coq proof assistant (Coq dev. team) — https://coq.inria.fr/

#13

Drapeau français (Wikipedia) — https://fr.m.wikipedia.org/wiki/Fichier:Flag_of_France.png

#17

G. van Rossum — https://medium.com/@cliffberg/the-creator-of-python-guido-van-rossum-recently-wrote-6f1053803385
https://www.zdnet.com/article/python-programming-language-creator-retires-saying-its-been-an-amazing-ride/

Sources : images, citations, liens III (avril 2024)

#18

Compcert https://compcert.org/

C99 standard https://www.dii.uchile.cl/~daespino/files/Iso_C_1999_definition.pdf
https://www.iso.org/fr/standard/29237.html

Langage OCaml https://ocaml.org/

#22

Frama-C https://www.frama-c.com/

Why3 https://www.why3.org/

alt-ergo (Solveur SMT) https://alt-ergo.ocamlpro.com/

#23

Toyota Camry https://en.wikipedia.org/wiki/2009%E2%80%932011_Toyota_vehicle_recalls

#24

ANSSI https://cyber.gouv.fr/
https://fr.wikipedia.org/wiki/Fichier:ANSSI_Logo.svg

Critères Communs https://cyber.gouv.fr/presentation-des-certifications-criteres-communs

#26

Méthode B http://www.atelierb.eu/
https://www.event-b.org/
1996 J.R. Abrial, The B Book, *Assigning Programs to Meanings*, Cambridge University Press

#29

TLA+ http://lamport.azurewebsites.net/tla/tla.html?from=https://research.microsoft.com/users/lamport/tla/tla.html&type=path
https://tla.msr-inria.inria.fr/tlaps/content/Home.html

#30

FoCaLiZe https://github.com/pessaux-f/focalize

Sources : images, citations, liens IV (avril 2024)

#33	
X. Yang et al.	*Finding and Understanding Bugs in C Compilers*, PLDI 2011, ACM SIGPLAN Notices 46(6), pp. 283–294. https://dl.acm.org/doi/proceedings/10.1145/1993498
#34	
HOL light	http://www.cl.cam.ac.uk/jrh13/hol-light/
Esterel	https://www.college-de-france.fr/fr/agenda/cours/esterel-de-z
#35	
Scade	https://www.ansys.com/products/embedded-software/ansys-scade-suite
Lustre	https://www-verimag.imag.fr/The-Lustre-Programming-Language-and?lang=en
	T. Bourke et al. *A Formally Verified Compiler for Lustre*, PLDI 2017, ACM, 586–601.https://inria.hal.science/hal-01512286/
#39	
Model-Checking	http://www.loria.fr/~merz/papers/mc-tutorial.pdf
	Prix Turing : https://amturing.acm.org/byyear.cfm
	SPIN (model-checker) : https://spinroot.com/
#43	
π-calcul	https://link.springer.com/referenceworkentry/10.1007/978-0-387-09766-4_202
PIPS	https://pips4u.org/
Sémantique parallélisme et multicore	A. Jade et al., *The semantics of power and ARM multiprocessor machine code*, 2009 Workshop on Declarative Aspects of Multicore Programming. 13–24. https://www.microsoft.com/en-us/research/publication/the-semantics-of-power-and-arm-multiprocessor-machine-code/
#45	
Interprétation abstraite	https://www.absint.com/
Verasco	http://compcert.inria.fr/verasco/
Astrée	https://www.absint.com/astree/index.htm
Polyspace	https://www.mathworks.com/products/polyspace.html
#46	
Précision des nombres flottants en Coq	http://coquelicot.saclay.inria.fr/description.html

Sources : images, citations, liens V (avril 2024)

#47

Anti-missiles patriot (af.mil)
https://www.afcent.af.mil/Units/455th-Air-Expeditionary-Wing/Photos/igphoto/2001477327/
https://veteransbreakfastclub.org/scuds-vs-patriots-desert-storm-1991/
https://www.cs.unc.edu/~smp/COMP205/LECTURES/ERROR/lec23/node4.html
Patriot missile defense: Software problems led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26, Information Management and Technology Division, US General Accounting Office, Washington DC, 1992.

F22 raptor (nationalinterest.org)
https://nationalinterest.org/blog/buzz/back-2016-f-22-raptor-lost-its-stealth-74816
https://www.itnews.com.au/news/stealth-fighters-hit-by-software-crash-74081

#49

Sceau CIA (Wikipedia)
https://fr.m.wikipedia.org/wiki/Fichier:Seal_of_the_Central_Intelligence_Agency_%28B%26W%29.svg

#50

Modèle de Dolev-Yao
https://members.loria.fr/VCortier/files/Teaching/tutorial.pdf

EasyCrypt
easycrypt.info

#51

Differential privacy
https://www.statice.ai/post/what-is-differential-privacy-definition-mechanisms-examples
https://privacytools.seas.harvard.edu/differential-privacy

G. Barthes et al., *Proving differential privacy in Hoare logic*, CSF 2014, IEEE, pp. 411–424. https://arxiv.org/abs/1407.2988

J. Hsu et al.
Differential Privacy: An Economic Method for Choosing Epsilon, CSF 2014, pp. 398–410. https://arxiv.org/abs/1402.3329

Traçabilité des données
https://dl.acm.org/doi/10.1145/3437992.3439920

Sources : images, citations, liens VI (avril 2024)

#52

Thales (Wikipedia)	https://fr.m.wikipedia.org/wiki/Fichier:Thales_Logo.svg
Belenios (vote éléctronique)	https://www.belenios.org/organiser-une-election.html
Cartes à puce (Gemalto, Trusted Logic, Thales)	https://capitalfinance.lesechos.fr/deals/sortie/gemalto-aux-commandes-de-trusted-logic-119261
	https://www.thalesgroup.com/en/markets/digital-identity-and-security
Passeport biométrique français (Wikipedia)	https://fr.wikipedia.org/wiki/Passeport_fran%C3%A7ais
	https://www.cnrs.fr/endirectdeslabos/Actualites/1759/Suite.aspx
	https://web.archive.org/web/20140829055608/http://www.cnrs.fr/ins2i/spip.php?article829
Protocoles	https://people.irisa.fr/Stephanie.Delaune/transparents/Annee2012/apmep-rouen.pdf
Politique de sécurité des aéroports	D. Delahaye et al., *A formal and sound transformation from Focal to UML : an application to airport security regulations*. Innov. Syst. Softw. Eng., 4(3), pp. 267–274, 2008. https://link.springer.com/article/10.1007/s11334-008-0060-5
Textes de loi	https://regmind.eu/presentation

#53

sel4	https://sel4.systems/
Sel4 is affordable	https://sel4.systems/Info/Docs/GD-NICTA-whitepaper.pdf
Microkernel	Q. Carbonneaux et al., *Applying Formal Verification to Microkernel IPC at Meta*, CPP 2022, pp. 116–129. https://dl.acm.org/doi/abs/10.1145/3497775.3503681

#54

BinSec	https://binsec.github.io/
Laboratoire Haute Sécurité	https://lhs.loria.fr/
	https://carbone.loria.fr/
Cybersécurité physique (offensive et autres)	https://www.serma-safety-security.com/
	https://www.nxp.com/applications/enabling-technologies/security:SECURITY-TECHNOLOGY
	https://www.qualcomm.com/company/corporate-responsibility/acting-responsibly/

Sources : images, citations, liens VII (avril 2024)

#56

D. Knuth https://www-cs-faculty.stanford.edu/~knuth/faq.html

#57

D. Hilbert (Oberwolfach Photo Collection) https://owpdb.mfo.de/detail?photo_id=12587