Qu'est-ce qu'on peut faire avec Coq et les méthodes formelles ?

Pierre Wilke

CentraleSupélec

mardi 23 octobre 2018

1. Formaliser les mathématiques
   - Théorème des quatre couleurs
   - Théorème de Feit-Thompson (de l'ordre impair)

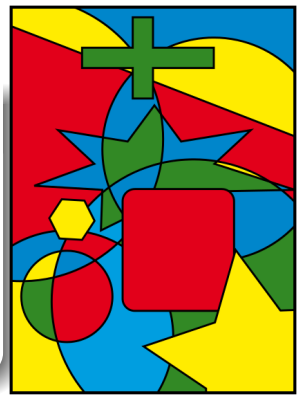
2. Raisonner sur des programmes
   - Sémantique
   - Compilation
   - Analyse statique
   - Preuves de programmes

Plan

Théorème des quatre couleurs

Théorème

Toute carte peut être coloriée avec au plus 4 couleurs, de sorte que deux régions adjacentes soient de couleurs différentes.

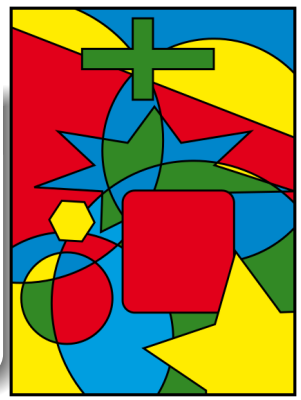


- Guthrie (1852) : conjecture
- Kempe (1879) : première preuve
- Heawood (1890) : découverte d'erreurs dans la preuve de Kempe
- Appel/Haken (1976) : première preuve par ordinateur
  - 1478 *cas critiques*
  - étudiés en 1200 heures !
  - **comment faire confiance à cette preuve ?**

Théorème des quatre couleurs

Théorème

Toute carte peut être coloriée avec au plus 4 couleurs, de sorte que deux régions adjacentes soient de couleurs différentes.



Gonthier / Werner (2015) : preuve en Coq (60 000 lignes)

On doit simplement vérifier l'énoncé du théorème.
Coq assure que la preuve est correcte.

Théorème de Feit-Thompson

Théorème

Tout groupe fini d'ordre impair est résoluble.

- Burnside (1911) : conjecture
- Feit-Thompson (1963) : preuve *compliquée* (250+ pages)
- Bender, Glauberman (1994) et Peterfalvi (2000) : simplification de la preuve (2 livres de 150+ pages chacun)
- comment vérifier l'exactitude de cette preuve ?

Gonthier (2012) : preuve en Coq

Plan

1. Formaliser les mathématiques

2. Raisonner sur des programmes
   - Sémantique
   - Compilation
   - Analyse statique
   - Preuves de programmes

Raisonnement sur des programmes

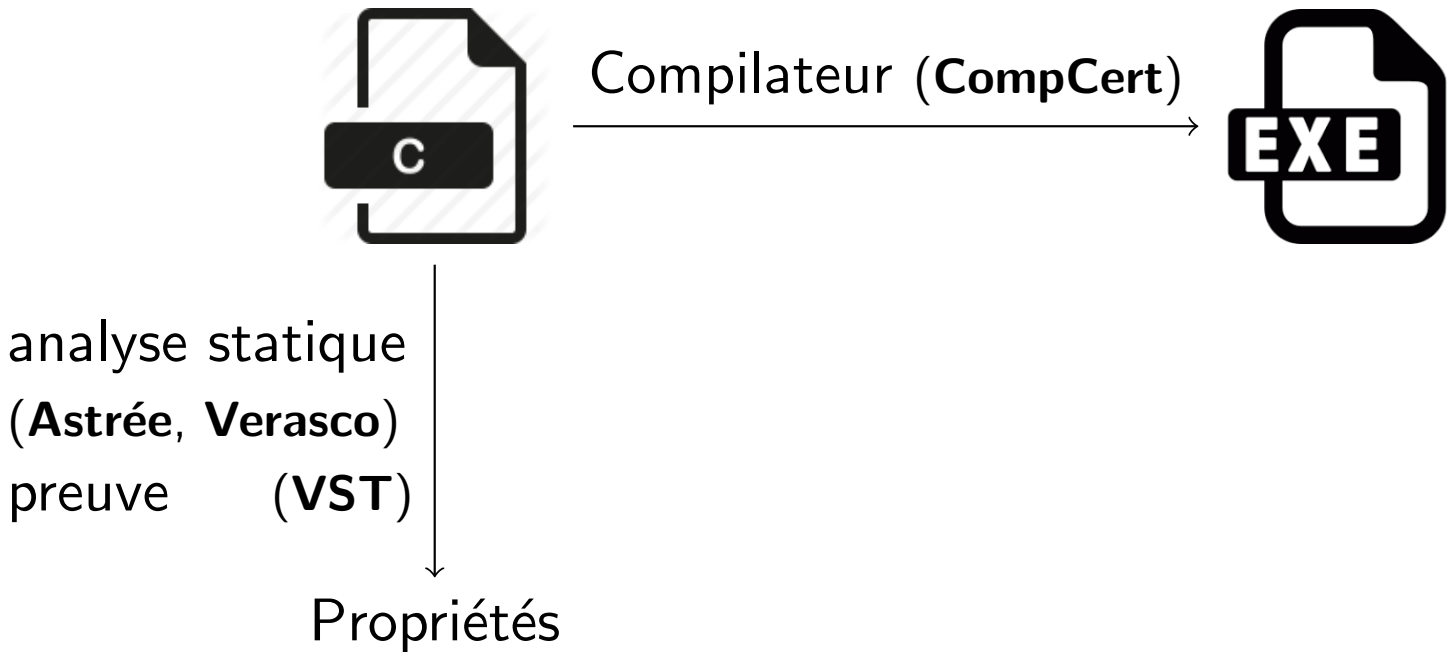
analyse statique
(**Astrée**, **Verasco**)
preuve (**VST**)

Propriétés

Exemples de propriétés intéressantes :

- le programme C ne provoque jamais de *buffer overflow*
- le programme C calcule $\sin(\frac{x\pi}{6})$

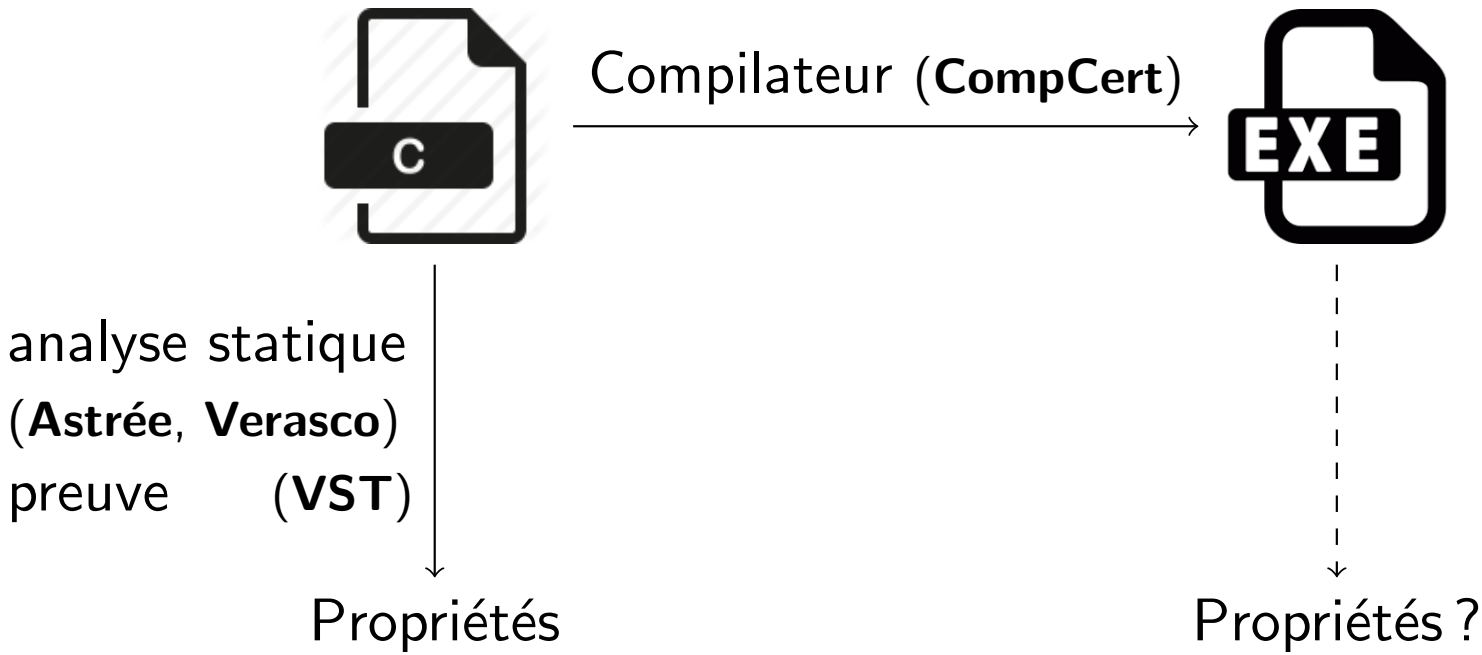
Raisonnement sur des programmes



Exemples de propriétés intéressantes :

- le programme C ne provoque jamais de *buffer overflow*
- le programme C calcule $\sin(\frac{x\pi}{6})$

Raisonnement sur des programmes



Exemples de propriétés intéressantes :

- le programme C ne provoque jamais de *buffer overflow*
- le programme C calcule $\sin(\frac{x\pi}{6})$

Sémantique formelle des langages de programmation

Objectif : associer une signification à la syntaxe des programmes

- quelle valeur est calculée par une expression donnée ?
- quel est l'effet d'une instruction sur l'état du programme ?
- que calcule ce programme ?

En TP, on a défini la sémantique des expressions avec `eval_expr`, et la sémantique des instructions du langage à pile avec `eval_instr` et `eval_prog`. L'état était composé de la pile.

Sémantique des expressions

La sémantique des expressions peut être donnée comme une fonction (comme en TP).

```
Fixpoint eval_expr (env: envt) (e: expr) : Z :=
match e with
| Const n => n
| Var v => env v
| Add e1 e2 => eval_expr env e1 + eval_expr env e2
| Mul e1 e2 => eval_expr env e1 * eval_expr env e2
| Sub e1 e2 => eval_expr env e1 - eval_expr env e2
end.
```

Le langage IMP

$$
\begin{array}{rcl}
s & ::= & \texttt{skip} \\
& | & x := e \\
& | & s_1 ; s_2 \\
& | & \texttt{if}(e)\ \texttt{then}\ s1\ \texttt{else}\ s2 \\
& | & \texttt{while}(e)\{s\}
\end{array}
$$

```
res := 1;
while (n != 0) {
  res := res * n;
  n := n - 1
}
```

À votre avis, quelle est la valeur contenue dans `res` en fonction de la valeur initiale de `n` ?

On ne peut pas donner la sémantique des programmes *via* une fonction : les programmes peuvent ne pas terminer.

Sémantique du langage IMP

On définit la sémantique de IMP comme la relation $\rightarrow$.

La formule $(s, e) \rightarrow e'$ signifie que l'instruction s transforme l'environnement e en e'.

Sémantique du langage IMP

$$\frac{}{(\texttt{skip}, env) \to env} \qquad \frac{}{(x{:=}e, env) \to \texttt{update\_env}\ env\ x\ (\texttt{eval\_expr}\ env\ e)}$$

$$\frac{(s_1, env) \to env' \qquad (s_2, env') \to env''}{(s_1; s_2, env) \to env''} \qquad \frac{\texttt{eval\_expr}\ env\ e = 0}{(\texttt{while}(e)\{s\}, env) \to env}$$

$$\frac{\texttt{eval\_expr}\ env\ e \neq 0 \qquad (s, env) \to env' \qquad (\texttt{while}(e)\{s\}, env') \to env''}{(\texttt{while}(e)\{s\}, env) \to env''}$$

$$\frac{\texttt{eval\_expr}\ env\ e \neq 0 \quad (s_1, env) \to env'}{(\texttt{if}(e)\ \texttt{then}\ s1\ \texttt{else}\ s2, env) \to env'} \qquad \frac{\texttt{eval\_expr}\ env\ e = 0 \quad (s_2, env) \to env'}{(\texttt{if}(e)\ \texttt{then}\ s1\ \texttt{else}\ s2, env) \to env'}$$

Sémantique du langage IMP

En Coq, on définirait la sémantique du langage IMP comme suit :

```
Inductive exec_prog : stmt * envt -> envt -> Prop :=
 | exec_prog_skip: forall env,
     exec_prog (skip, env) env
 | exec_prog_assign: forall env x e,
     exec_prog (Assign x e, env)
      (update_env env x (eval_expr env e))
 | exec_prog_seq: forall env1 env2 env3 s1 s2,
     exec_prog (s1, env1) env2 ->
     exec_prog (s2, env2) env3 ->
     exec_prog (Seq s1 s2, env1) env3
 | (* ... *).
```

Il s'agit d'un prédicat inductif, qui énumère les différentes manières de prouver `exec_prog (s,e) e'`.

Un compilateur C formellement prouvé : CompCert

CompCert (`http://compcert.inria.fr/`, depuis 2005)

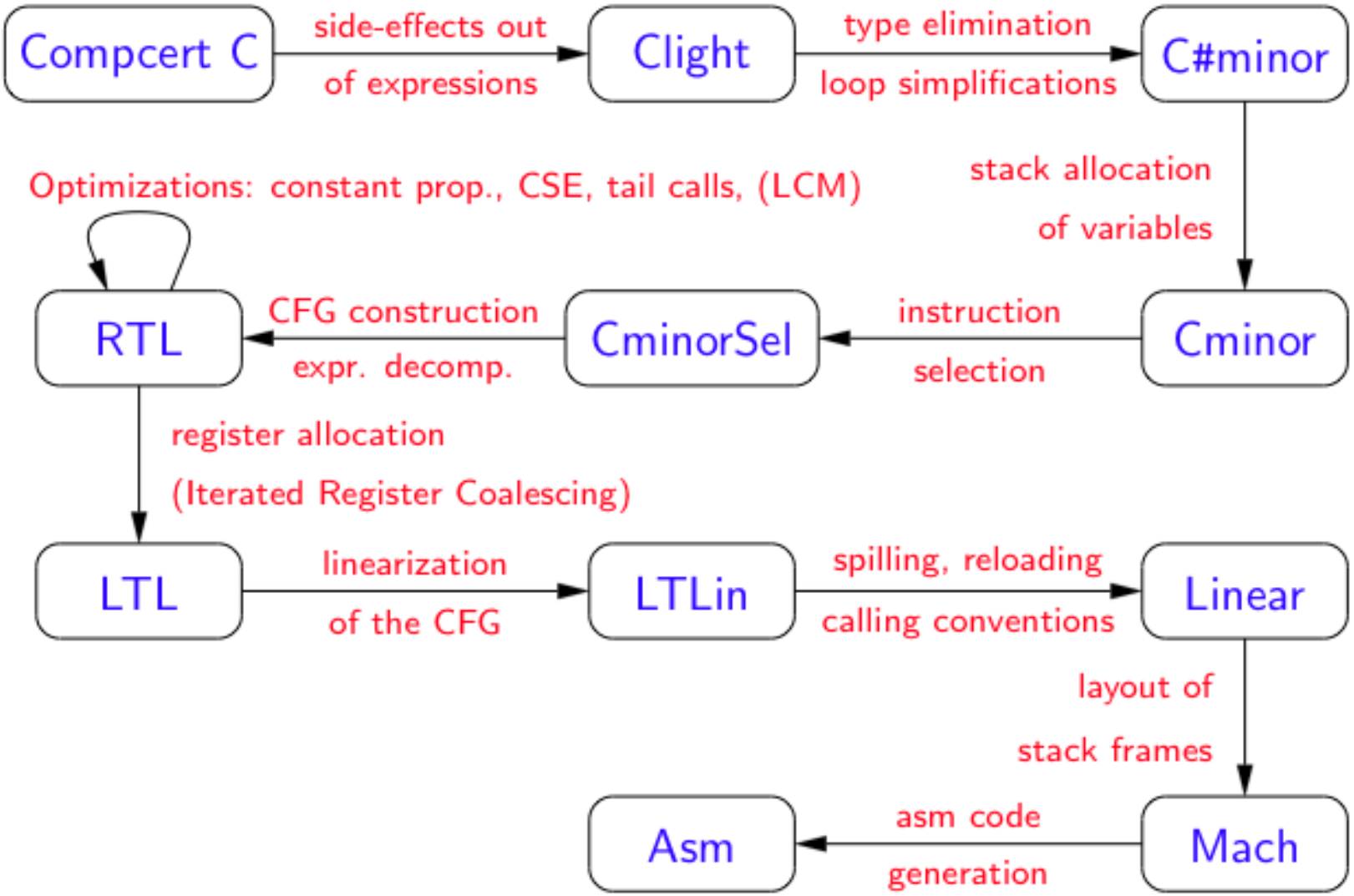
- compilateur (pour C) écrit en Coq par X. Leroy et S. Blazy
- pour les architectures x86, ARM, PowerPC, RISC-V
- définition de **sémantiques formelles** pour C et assembleur
- **preuve de correction** du compilateur

Utilisateurs industriels : MTU (nucléaire allemand), Airbus, ...

Théorème (Correction du compilateur)

Si le programme P_C ne contient pas de comportement indéfini, et si CompCert produit un programme assembleur P_{Asm} sans erreurs, alors P_{Asm} a le même comportement que P_C.

Passes de compilation de CompCert



La sémantique du langage C

Documents *informels* décrivant le comportement attendu des programmes C

- The C programming language (Kerninghan & Ritchie, 1978)
- ANSI C (1989)
- ISO C90, C95, C99, C11, C18

Problèmes :

- documents *informels*
- subtilités : valeurs indéterminées, **comportements indéfinis**...

Comportements indéfinis en C

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

standard C11

exemples de comportements indéfinis :

- accès à un pointeur nul
- accès hors des bornes d'un tableau
- accès à des variables non-initialisées
- + une centaine de cas décrits dans le standard...

Sémantique formelle de C et assembleur dans CompCert

État d'un programme C :

- état de la mémoire
- environnement pour les variables locales
- environnement pour les variables globales
- reste du programme à évaluer

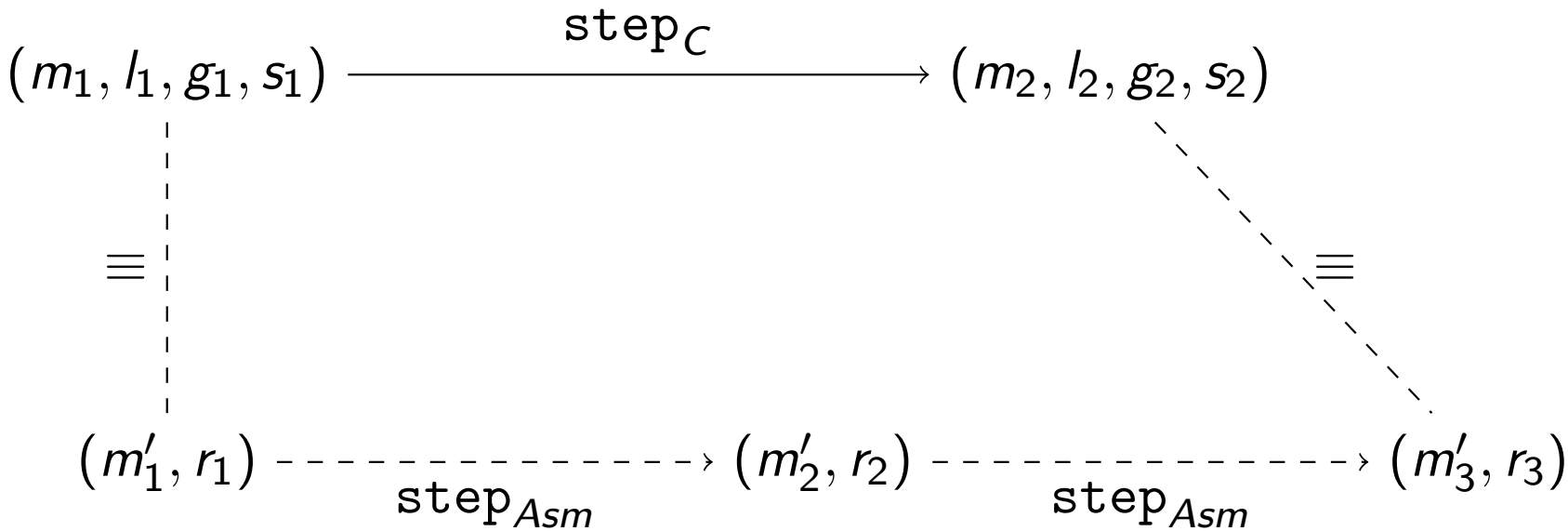
État d'un programme assembleur :

- état de la mémoire
- environnement pour les registres

Sémantique d'un programme Asm :

- `step` : transitions d'un état à un autre en fonction de l'instruction courante
- clôture réflexive transitive de `step` : 0, 1, ou plusieurs étapes

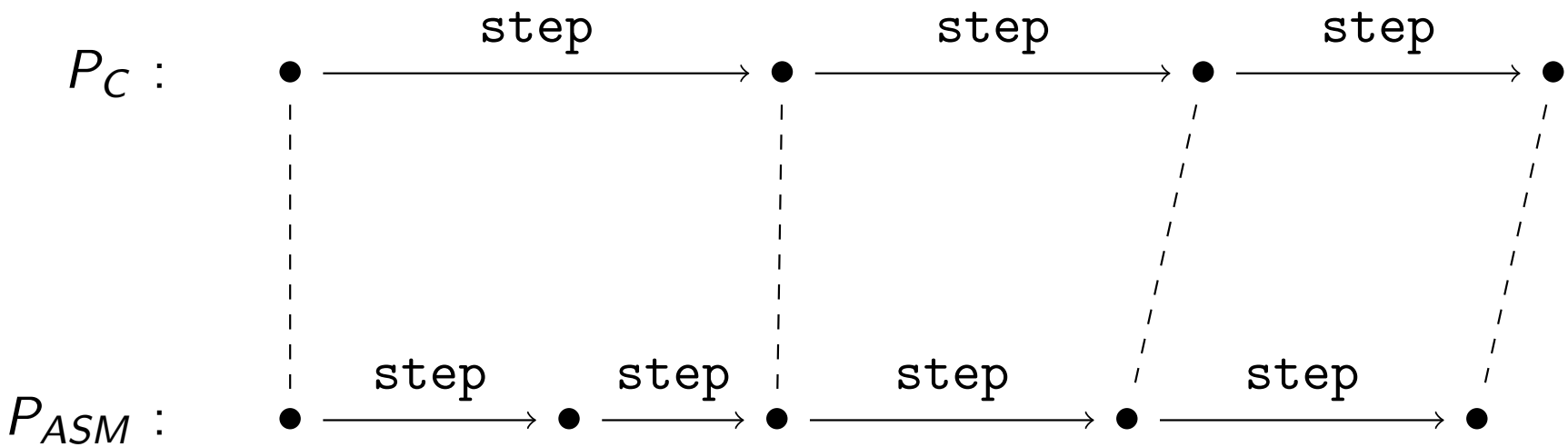
Preuve de correction de la compilation : simulation



Pour tous états C et assembleur *équivalents*, pour chaque pas au niveau C, il existe une séquence de pas au niveau assembleur, telle que les états résultants sont équivalents.

États équivalents : chaque emplacement mémoire en C a un emplacement mémoire correspondant au niveau assembleur.

Preuve de correction de la compilation : simulation



Pour chaque pas dans le programme C, en partant d'états C et assembleur *équivalents*, il existe un certain nombre de pas assembleur, tel que les états résultants sont équivalents.

Au final : tout comportement au niveau C a une contrepartie (un comportement équivalent) au niveau assembleur.

Absence de bugs dans CompCert

> The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users. [13]

Finding and understanding bugs in C compilers. PLDI'11

But de l'analyse statique

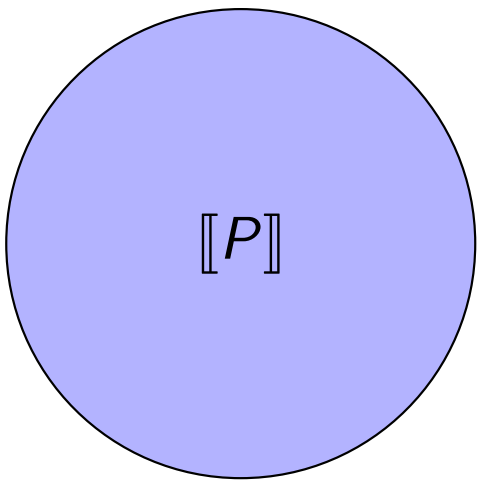
Objectif : prouver que tous les états qu'un programme peut atteindre sont sûrs

Un (vieil) exemple : Ariane 5 (1996) [1]
La fusée a explosé 36,7 secondes après le décollage.
Cause du bug : dépassement d'entier (*integer overflow*
Les valeurs reçues par l'accéléromètre étaient trop importantes pour la capacité des entiers, ce qui a entraîné des commandes de guidage erronées.

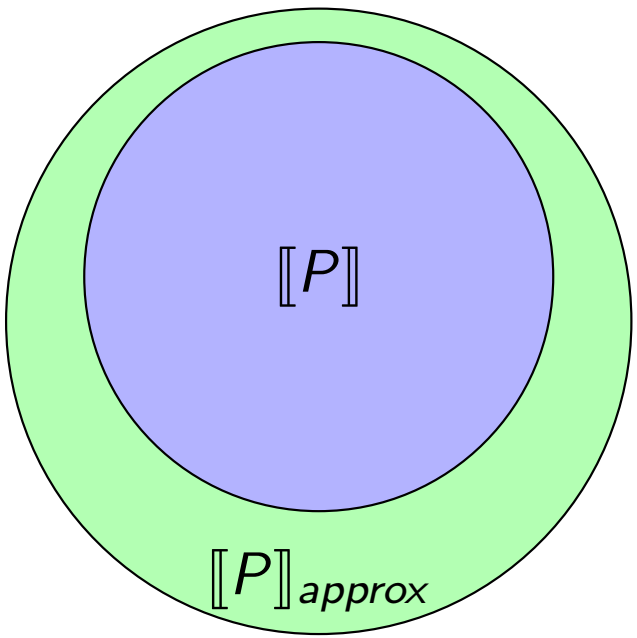
On peut prouver par analyse statique l'absence de ces dépassements d'entier.

1. `https://fr.wikipedia.org/wiki/Vol_501_d'Ariane_5`

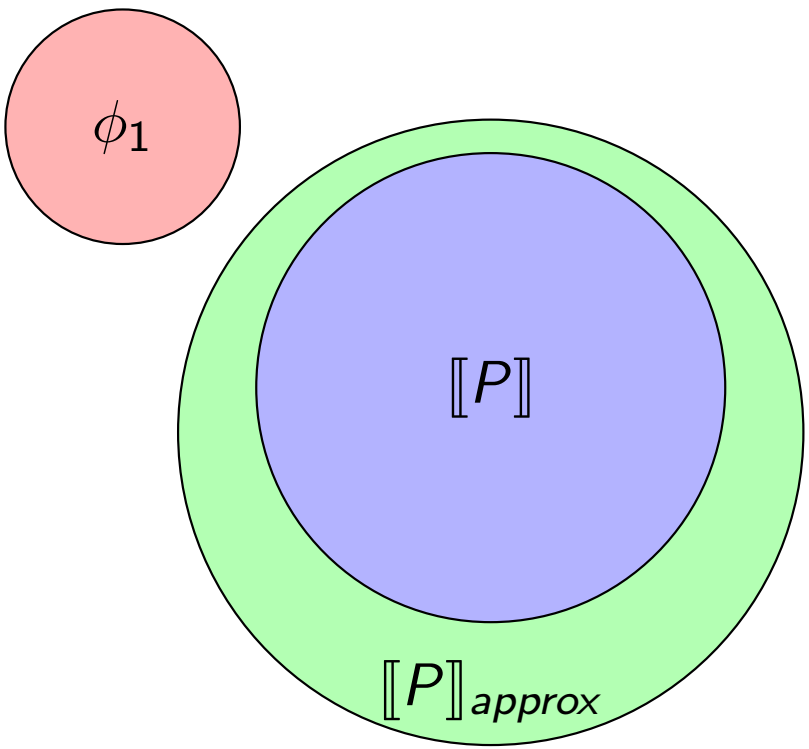
Analyse statique : principes



Analyse statique : principes

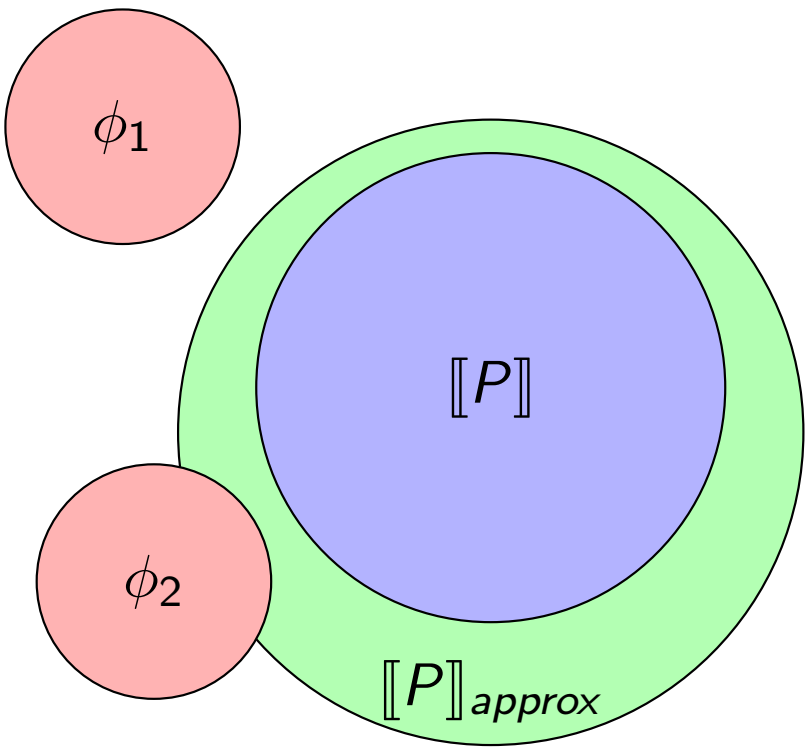


Analyse statique : principes



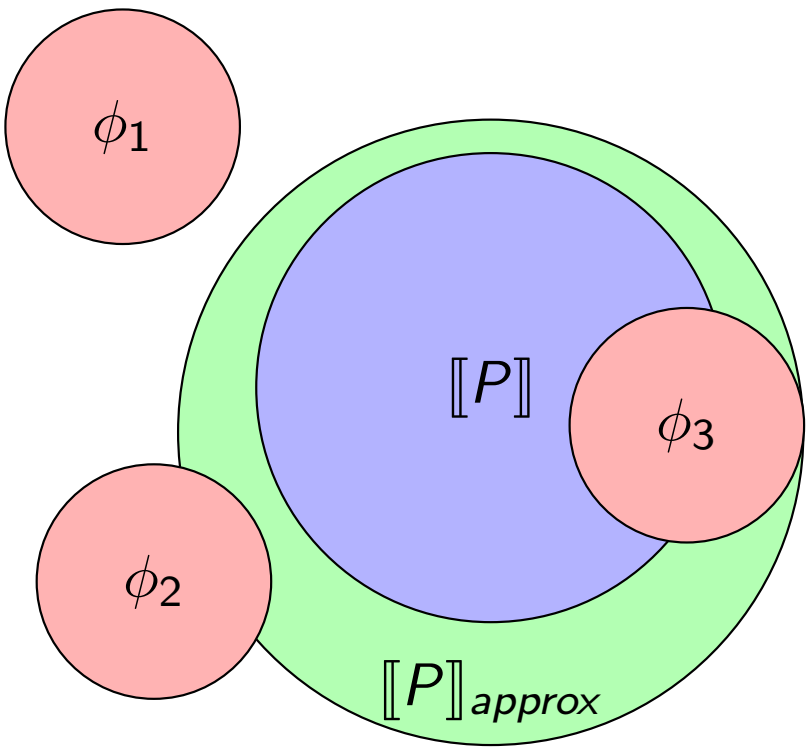
- P est sûr par rapport à ϕ_1 et l'analyse le prouve $[\![P]\!]_{approx} \cap \phi_1 = \emptyset$

Analyse statique : principes



- P est sûr par rapport à ϕ_1 et l'analyse le prouve
  $[\![P]\!]_{approx} \cap \phi_1 = \emptyset$
- P est sûr par rapport à ϕ_2 mais l'analyse ne le prouve pas : fausse alerte
  $[\![P]\!]_{approx} \cap \phi_2 \neq \emptyset$

Analyse statique : principes



- P est sûr par rapport à ϕ_1 et l'analyse le prouve
  $[\![P]\!]_{approx} \cap \phi_1 = \emptyset$
- P est sûr par rapport à ϕ_2 mais l'analyse ne le prouve pas : fausse alerte
  $[\![P]\!]_{approx} \cap \phi_2 \neq \emptyset$
- P n'est pas sûr par rapport à ϕ_3 et l'analyse le prouve
  $[\![P]\!]_{approx} \cap \phi_3 \neq \emptyset$

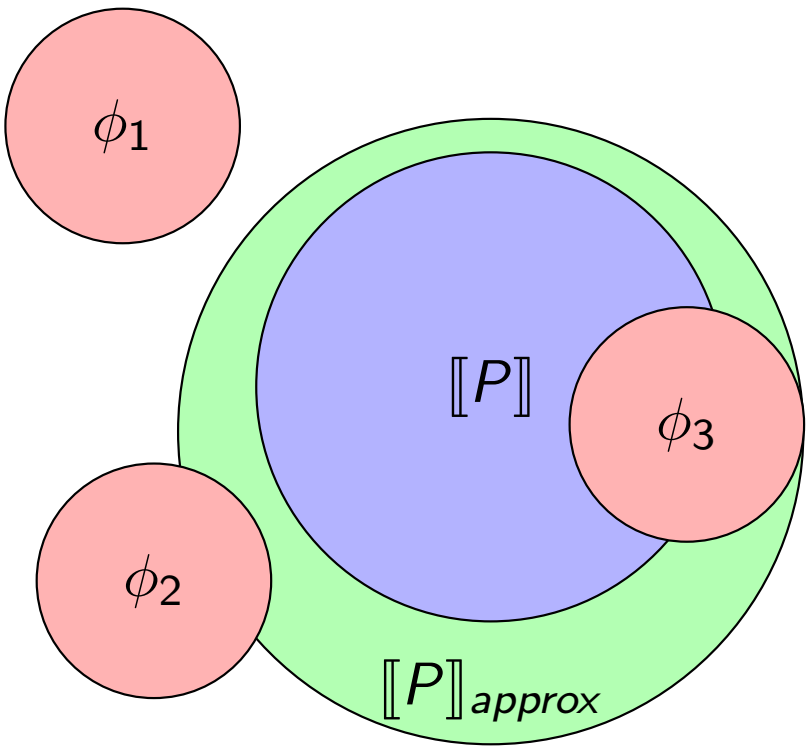
Analyse statique : principes

Comme on ne peut pas calculer exactement $[\![P]\!]$, on en calcule une sur-approximation $[\![P]\!]_{approx}$

Sur-approximation :
$[\![P]\!] \subseteq [\![P]\!]_{approx}$

Si $[\![P]\!]$ a des comportements indésirables (c'est-à-dire dans ϕ), alors $[\![P]\!]_{approx}$ aussi :

$$[\![P]\!] \cap \phi \neq \emptyset \Rightarrow [\![P]\!]_{approx} \cap \phi \neq \emptyset$$



Analyses statiques dans le vrai monde

Astrée (`https://www.absint.com/astree/index_fr.htm`)

Astrée a été utilisé pour vérifier l'absence d'erreurs à l'exécution dans le logiciel de commande de vol électrique primaire de deux types d'avion Airbus.

AIRBUS

Verasco (`http://verasco.imag.fr`)
Analyse statique formellement vérifiée (en Coq) qui vérifie qu'un programme P écrit en C n'a pas de comportements indéfinis. Si l'analyse réussit, le programme ne peut pas planter !

Preuve de programmes

```
res := 1;
while (n != 0) {
  res := res * n;
  n := n - 1
}
```

Que fait ce programme ?

Preuve de programmes

```
res := 1;
while (n != 0) {
  res := res * n;
  n := n - 1
}
```

Que fait ce programme ? Il calcule $n!$.
Comment en être sûr ?

Triplet de Hoare

On considère des triplets de Hoare $\{P\}s\{Q\}$, où :

- P et Q sont des formules logiques sur l'état du programme
- s est une instruction du langage IMP

Signification d'un triplet $\{P\}s\{Q\}$

Si la précondition P est vérifiée avant l'exécution de s, alors la postcondition Q est vérifiée après son exécution.

Exemples (valides ou pas)

- $\{\texttt{True}\}x{:=}3\{x = 3\}$
- $\{x > 0\}x{:=}x + 1\{x > 0\}$
- $\{x > 0\}x{:=}x - 1\{x < 2\}$
- $\{x = n\}x{:=}x + 1\{x = n + 1\}$
- $\{True\}\texttt{if } (x < 0) \texttt{ then } x{:=}0 - x \texttt{ else skip}\{x > 0\}$

Validité des triplets : lien avec la sémantique

Validité d'un triplet $\{P\}s\{Q\}$

Un tel triplet est valide si :
en partant de n'importe quel état *env* qui vérifie P,
si l'exécution de P aboutit à l'état env' $((P, env) \to env')$,
alors env' vérifie Q.

On dispose d'un certain nombre de règles pour prouver que des triplets de Hoare sont valides.
Ces règles forment ce qu'on appelle la **logique de Hoare**

Logique de Hoare : séquence

$$\frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

Exemple

$$\frac{\{x > 0\}x{:=}x + 1\{x > 1\} \qquad \{x > 1\}y{:=}2 * x\{y > 2\}}{\{x > 0\}x{:=}x + 1; y{:=}2 * x\{y > 2\}}$$

Logique de Hoare : conditionnelle

$$\frac{\{P \wedge \texttt{eval\_expr}\ e = \texttt{vrai}\}s_1\{Q\} \quad \{P \wedge \texttt{eval\_expr}\ e = \texttt{faux}\}s_2\{Q\}}{\{P\}\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\{Q\}}$$

Exemple

$$\frac{\{\texttt{True} \wedge y \geq 0\}x{:=}y\{x \geq 0\} \qquad \{\texttt{True} \wedge y < 0\}x{:=}-y\{x \geq 0\}}{\{\texttt{True}\}\texttt{if}\ (y \geq 0)\ \texttt{then}\ x{:=}y\ \texttt{else}\ x{:=}-y\{x \geq 0\}}$$

Logique de Hoare : affectation

$$\overline{\{P[x \leftarrow E]\}\ x{:=}E\ \{P\}}$$

Si le prédicat P, en remplaçant x par E, est vrai avant l'exécution de l'affectation, alors le prédicat P est vrai après.

Logique de Hoare : while

$$\frac{\{P \wedge \texttt{eval\_expr}\ e = \texttt{vrai}\}s\{P\}}{\{P\}\ \texttt{while}(e)\{s\}\ \{P \wedge \texttt{eval\_expr}\ e = \texttt{faux}\}}$$

- P : invariant de boucle

Logique de Hoare

Renforcement de la précondition et affaiblissement de la postcondition

$$\frac{P' \Rightarrow P \qquad \{P\}s\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}s\{Q'\}}$$

Récapitulatif des règles

$$\textsc{weaken}\ \frac{P' \Rightarrow P \qquad \{P\}s\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}s\{Q'\}}$$

$$\textsc{séquence}\ \frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

$$\textsc{while}\ \frac{\{P \wedge e\}s\{P\}}{\{P\}\ \texttt{while}(e)\{s\}\ \{P \wedge \neg e\}}$$

$$\textsc{affectation}\ \frac{}{\{P[x \leftarrow E]\}\ x{:=}E\ \{P\}}$$

$$\textsc{condition}\ \frac{\{P \wedge e\}s_1\{Q\} \qquad \{P \wedge \neg e\}s_2\{Q\}}{\{P\}\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\{Q\}}$$

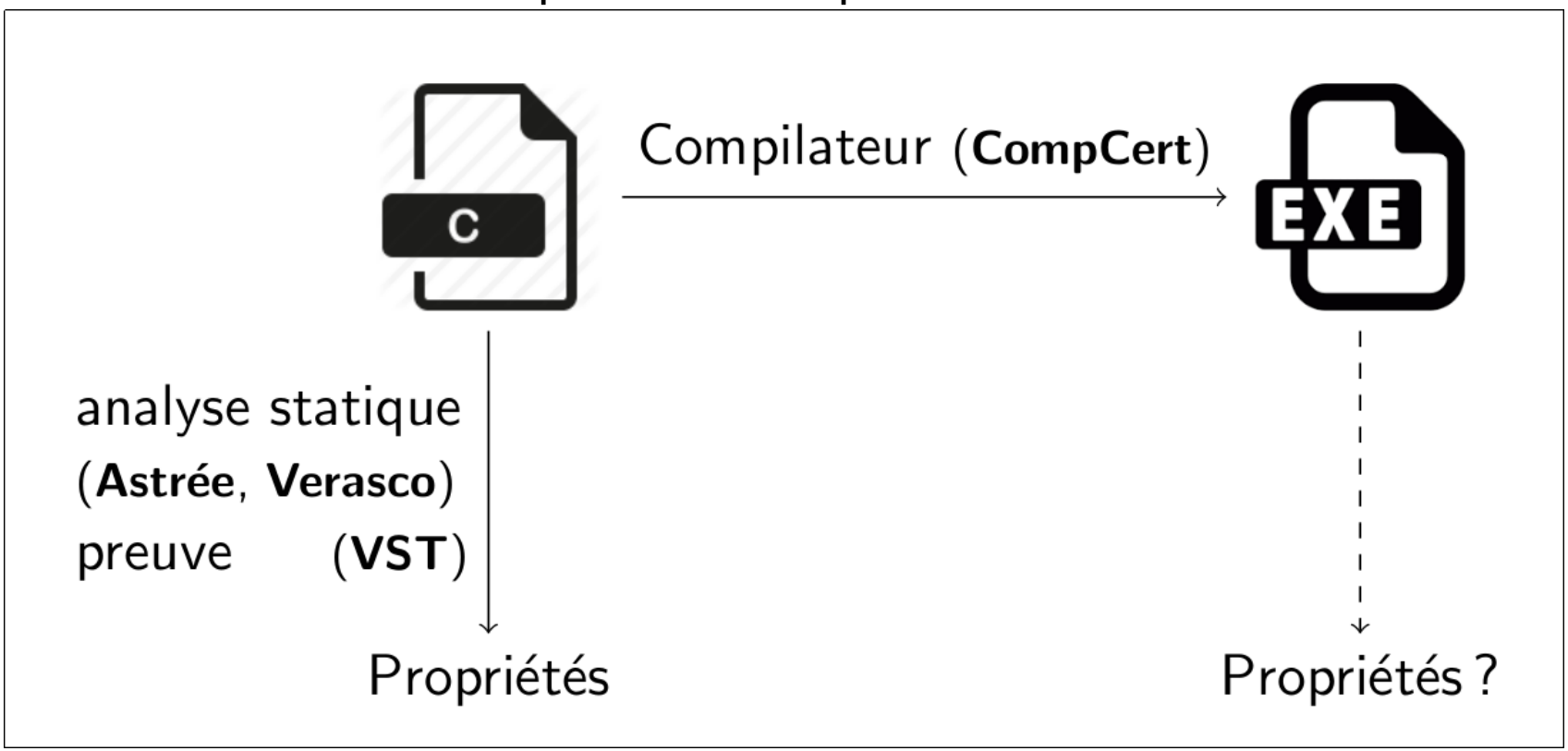
Logique de Hoare : exemple

$$\frac{\dots}{\{x \leq b\}(\texttt{while}\ (x < b)\{x := x + 1\})\ \{x = b\}}$$

Logique de Hoare dans la vraie vie

Verifiable Software Toolchain (`http://vst.cs.princeton.edu/`)

- preuves de programmes sur des programmes C
- Connexion avec le compilateur CompCert



Un système d'exploitation formellement vérifié : CertiKOS

CertiKOS (`http://flint.cs.yale.edu/certikos/`)

Système d'exploitation : gestion de la mémoire, des processus, des périphériques, du système de fichiers *via des appels système*

On peut *spécifier* le comportement attendu de chaque appel système.

Le code du système d'exploitation doit satisfaire cette spécification : **preuve de programmes.**

Le code compilé doit aussi satisfaire la spécification : **CompCert**.