# Compilation vérifiée : vers du logiciel zéro défaut

Sandrine Blazy



Sciences du logiciel : de l'idée au binaire, INS2I, 2022-09-20

# Note

Certaines diapositives sont reprises d'un cours de Xavier Leroy.


https://www.college-de-france.fr/agenda/cours/semantiques-mecanisees-quand-la-machine-raisonne-sur-ses-langages

# À quels risques sont exposés les compilateurs ?

Produire du code exécutable faux à partir d'un programme source correct

> We found and reported **hundreds** of previously **unknown** bugs [...]. Many of the bugs we found cause a compiler to emit incorrect code **without any warning**. 25 of the bugs we reported against GCC were classified as **release-blocking**.

[Yang, Chen, Eide, Regehr. Finding and understanding bugs in C compilers. PLDI'11]

# Compilation vérifiée

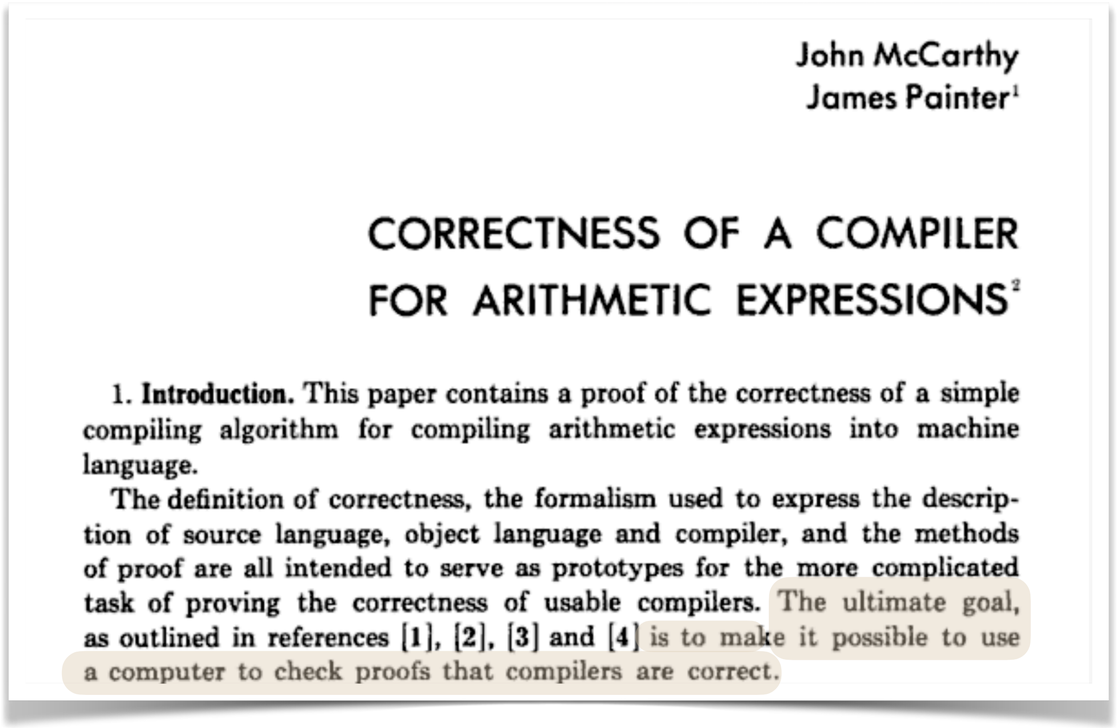
Les compilateurs sont des programmes complexes, mais ont une spécification « de bout en bout » assez simple :

> Le code produit doit s'exécuter comme prescrit par la sémantique du programme source.

Cette spécification devient mathématiquement précise dès que l'on dispose de sémantiques formelles pour les langages source et cible.

Dès lors on peut envisager de vérifier le compilateur.

# Une idée ancienne …



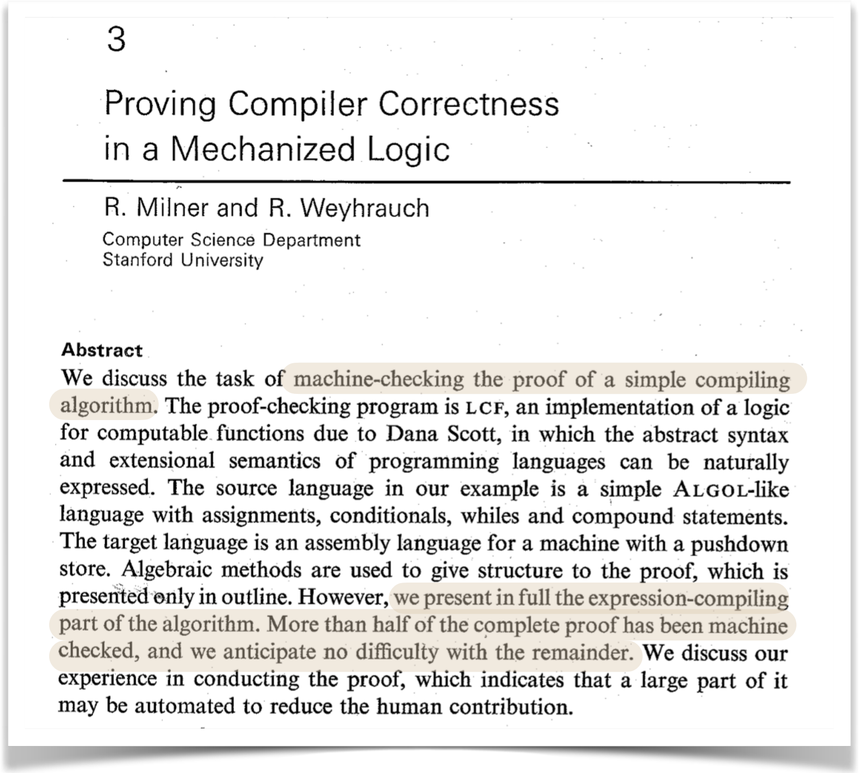
John McCarthy
James Painter[1]

## CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

1. **Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967



3

## Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

**Abstract**

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Machine Intelligence (7), 1972

# Dorénavant enseignée en master

(Sémantiques mécanisées : quand la machine raisonne sur ses langages, X.Leroy)
(Software foundations, B.Pierce et al.)

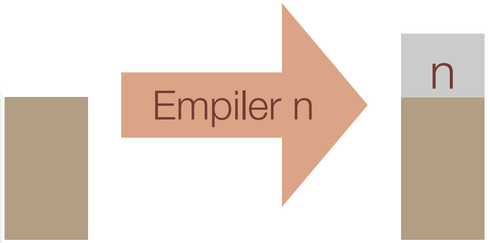
```
type exp = Nb of int | Id of string | Plus of exp * exp
```

```
type etat = string → int
```

```
let rec eval (e:etat)(a:exp): int =
match a with
   | Nb n → n
   | Id x → e x
   | Plus (a1,a2) → (eval e a1)+(eval e a2)
```


```
type instr = Empiler of int | Lire of string   | IPlus
```

```
let rec exec (e:etat)(pile: int list)(pgm: instr list): int list =
  match (pgm, pile) with
   | ([], _) → pile
   | (Empiler n :: pgm', _) → exec e (n :: pile) pgm'
   | (Lire x :: pgm', _) → exec e (e x :: pile) pgm'
   | (IPlus :: pgm', n:: m :: pile') → exec e ((m+n) :: pile') pgm'
   | (_ :: pgm', _) → exec e pile pgm'
```



# Dorénavant enseignée en master

(Sémantiques mécanisées : quand la machine raisonne sur ses langages, X.Leroy)
(Software foundations, B.Pierce et al.)

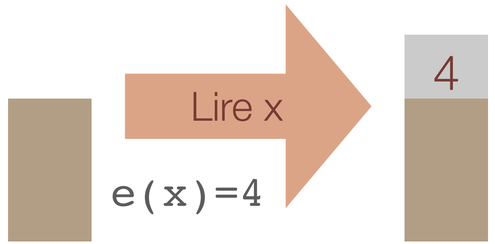
```
type exp = Nb of int | Id of string | Plus of exp * exp
```

```
type etat = string → int
```

```
let rec eval (e:etat)(a:exp): int =
match a with
   | Nb n → n
   | Id x → e x
   | Plus (a1,a2) → (eval e a1)+(eval e a2)
```


```
type instr = Empiler of int | Lire of string   | IPlus
```

```
let rec exec (e:etat)(pile: int list)(pgm: instr list): int list =
  match (pgm, pile) with
   | ([], _) → pile
   | (Empiler n :: pgm', _) → exec e (n :: pile) pgm'
   | (Lire x :: pgm', _) → exec e (e x :: pile) pgm'
   | (IPlus :: pgm', n:: m :: pile') → exec e ((m+n) :: pile') pgm'
   | (_ :: pgm', _) → exec e pile pgm'
```



# Dorénavant enseignée en master

(Sémantiques mécanisées : quand la machine raisonne sur ses langages, X.Leroy)
(Software foundations, B.Pierce et al.)

```
type exp = Nb of int | Id of string | Plus of exp * exp
```

```
type etat = string → int
```

```
let rec eval (e:etat)(a:exp): int =
match a with
   | Nb n → n
   | Id x → e x
   | Plus (a1,a2) → (eval e a1)+(eval e a2)
```


```
type instr = Empiler of int | Lire of string   | IPlus
```

```
let rec exec (e:etat)(pile: int list)(pgm: instr list): int list =
  match (pgm, pile) with
   | ([], _) → pile
   | (Empiler n :: pgm', _) → exec e (n :: pile) pgm'
   | (Lire x :: pgm', _) → exec e (e x :: pile) pgm'
   | (IPlus :: pgm', n:: m :: pile') → exec e ((m+n) :: pile') pgm'
   | (_ :: pgm', _) → exec e pile pgm'
```

# Dorénavant enseignée en master

(Sémantiques mécanisées : quand la machine raisonne sur ses langages, X.Leroy)
(Software foundations, B.Pierce et al.)

```
type exp = Nb of int | Id of string | Plus of exp * exp
```

```
type etat = string → int
```

```
let rec eval (e:etat)(a:exp): int =
match a with
   | Nb n → n
   | Id x → e x
   | Plus (a1,a2) → (eval e a1)+(eval e
```

compilation

sémantiques (eval, exec)

compilateur (compile)

```
let rec compile (a:exp): instr list =  match a with
   | Nb n →  [ Empiler n ]
   | Id x →  [ Lire x ]
   | Plus (a1,a2) → (compile a1)@ (compile a2)@ [IPlus]
```


```
type instr = Empiler of int | Lire of string  | IPlus
```

```
let rec exec (e:etat)(pile: int list)(pgm: instr list): int list =
  match (pgm, pile) with
   | ([], _) → pile
   | (Empiler n :: pgm', _) → exec e (n :: pile) pgm'
   | (Lire x :: pgm', _) → exec e (e x :: pile) pgm'
   | (IPlus :: pgm', n:: m :: pile') → exec e ((m+n) :: pile') pgm'
   | (_ :: pgm', _) → exec e pile pgm'
```

# Démontrer une propriété avec le logiciel Coq

ACM SIGPLAN Programming Languages Software award 2013
ACM Software System award 2013

coq.inria.fr

```
Theorem petit-compilateur-correct:
  forall e a,
  exec e [] (compile a) = [eval e a].
Proof.
  intros;
  … (* à compléter *)
Qed.
```

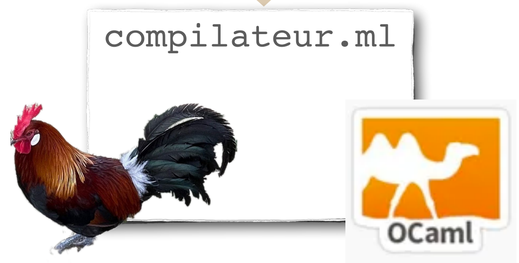
```
Extraction compile.
```

sémantiques (**eval**, **exec**)

compilateur (**compile**)

démonstration guidée par le logiciel Coq

extraction



Comment passer d’un prototype jouet conçu dans un laboratoire à un compilateur pour de vrais langages et de vraies architectures ?
(CompCert, CakeML)

# Le compilateur formellement vérifié CompCert

(X.Leroy, S.Blazy et al.) compcert.org

---

Compilateur modérément optimisant du langage C

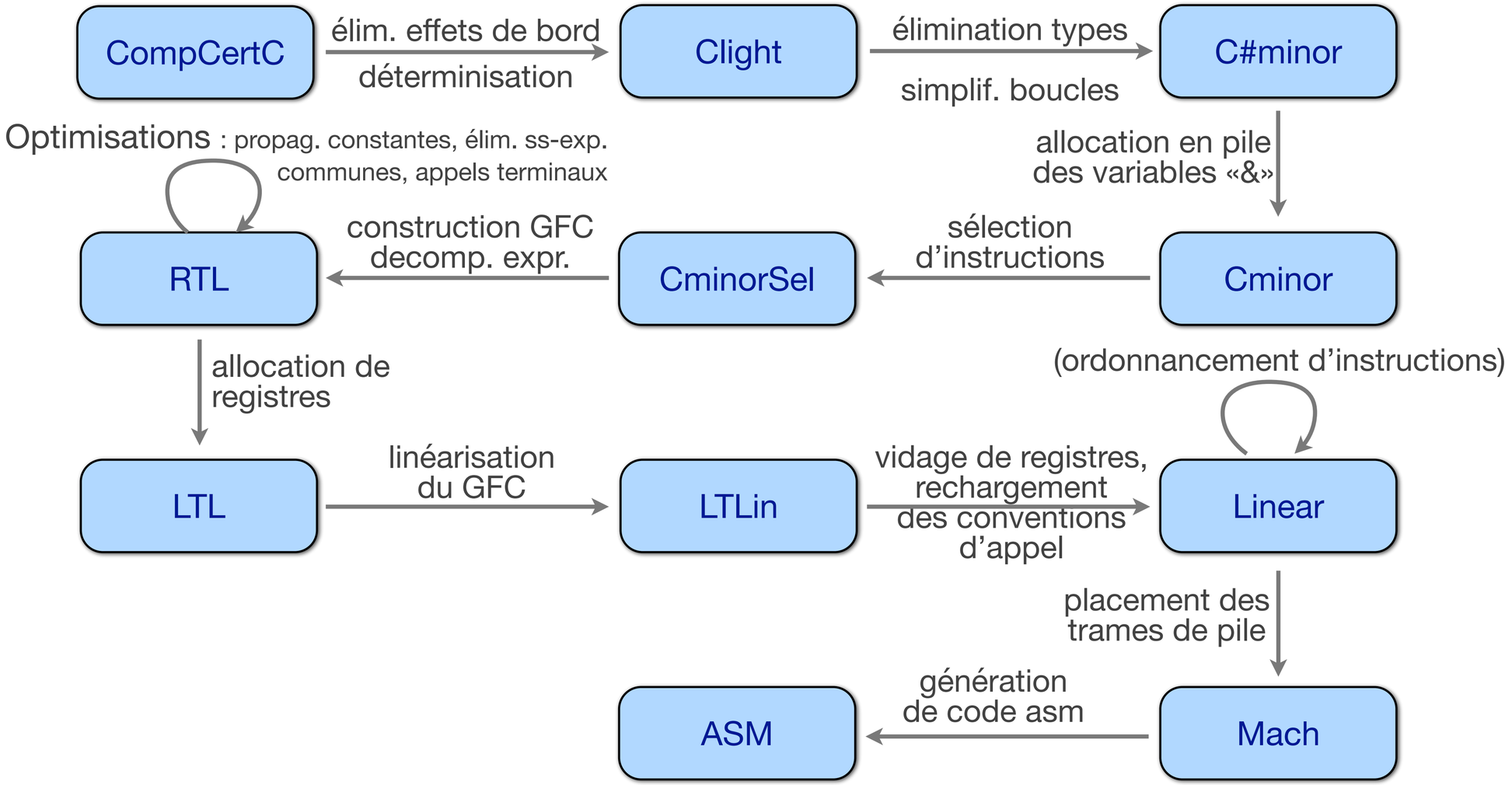
Cible les architectures les plus répandues (x86, ARM, PowerPC, RISC-V)

Programmé et vérifié à l'aide de l'assistant à la démonstration Coq

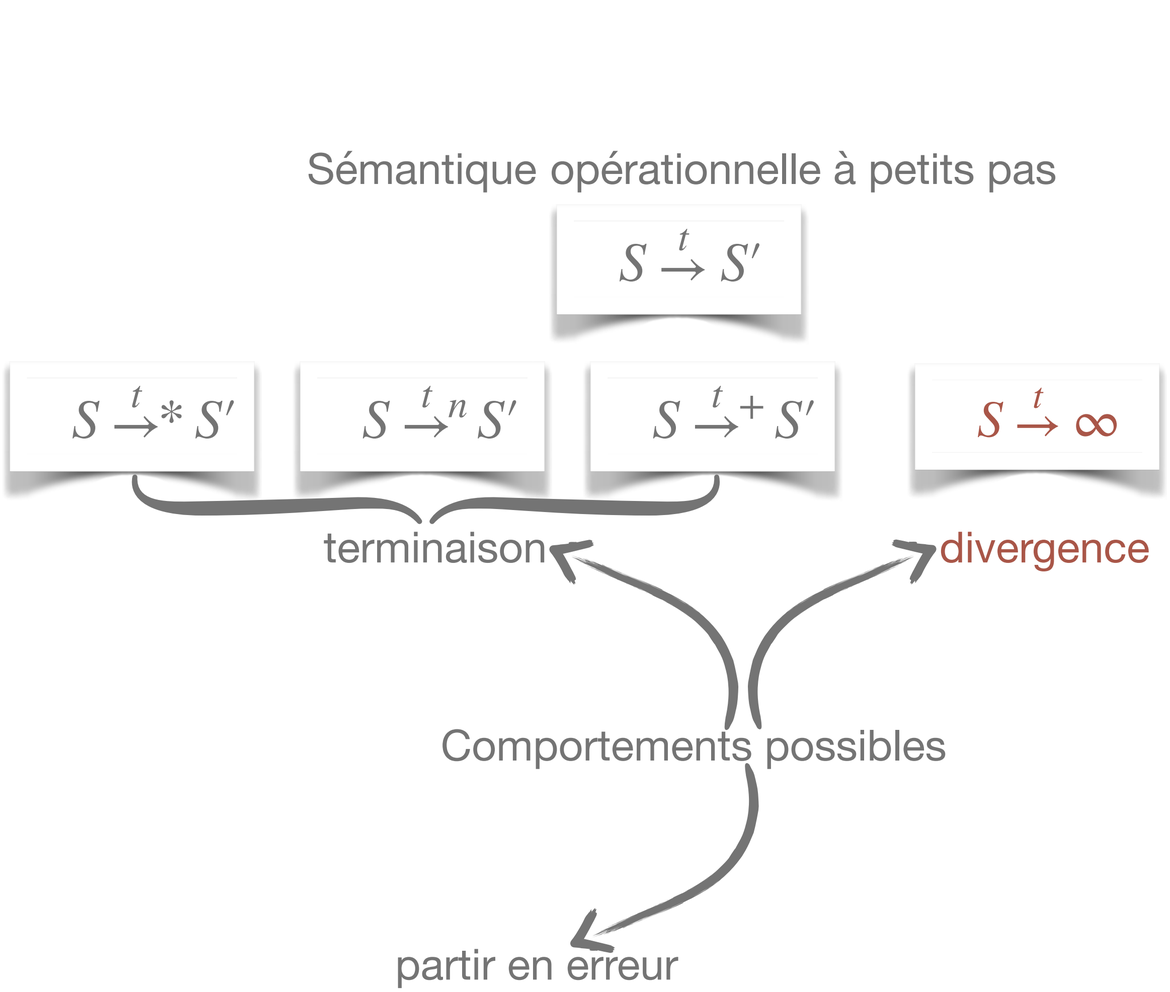
Commercialisé par AbsInt
Utilisé dans l'industrie pour du logiciel embarqué critique (nucléaire, avionique) ayant été qualifié IEC 60880 (nucléaire) : amélioration des performances du code généré, tout en garantissant des exigences de traçabilité

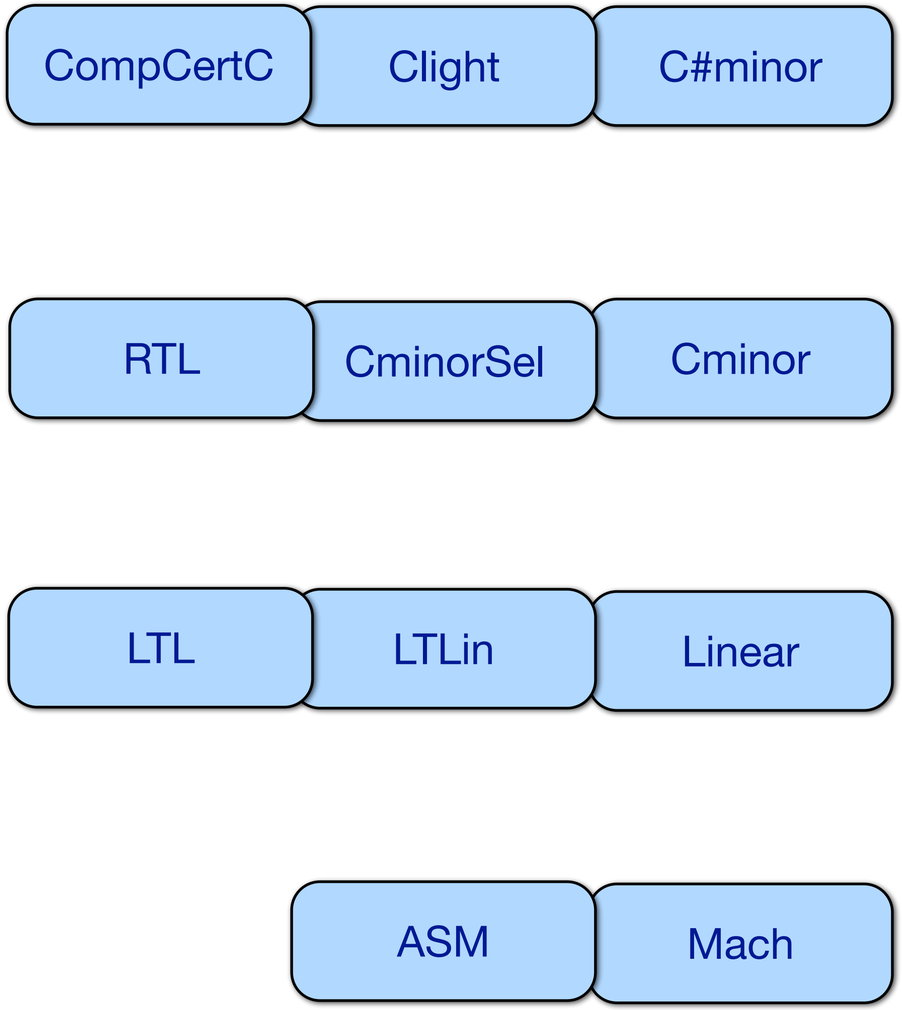
ACM SIGPLAN Programming Languages Software award 2022
ACM Software System award 2021

# Compilateur CompCert : 11 langages, 18 passes



# Compilateur CompCert : 11 langages, 18 passes



# Correction du compilateur : propriété de préservation sémantique

sémantiques (**exec_CompCertC**, **exec_ASM**)

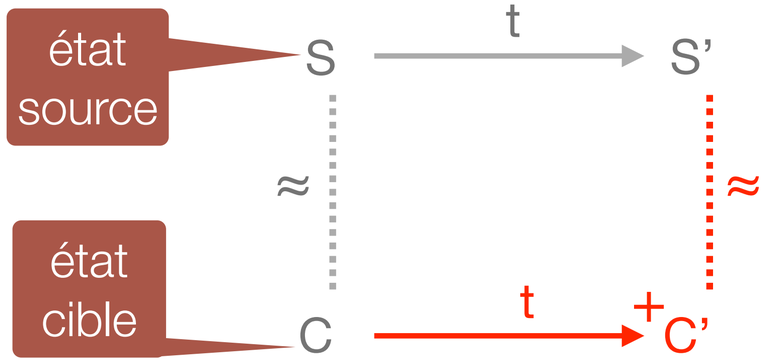
compilateur (**compile**)

Comportements préservés = terminaison et divergence

```
Theorem compilateur-correct:
  forall S C b,
  compile S = OK C →
  exec_CompCertC S b →
  exec_ASM C b.
```

« Le code produit C doit s'exécuter comme prescrit par la sémantique du programme source S. »

Technique de démonstration : diagrammes de simulation




démonstration guidée par le logiciel Coq

# Compilateur vérifié CompCert : principaux ingrédients

| Compilation | Sémantiques formelles | Vérification déductive |
| --- | --- | --- |
| Langages source et cible | Comportements observables | Assistant à la démonstration |
| **Langages intermédiaires** | **Traces d'événements externes d'E/S** | Théorème de préservation sémantique |
| **Optimisations** | **Petits pas** | **Diagramme de simulation** |
| **Analyses flot de données** | **Continuations** | **Mesure anti-bégaiement** |
| **Allocation de registres** | **Modèle mémoire** | **Validation a posteriori** |
| **Autres passes** | | |

# CT-CompCert, un compilateur sécurisé contre les attaques par canaux cachés temporels

Discipline de programmation « temps constant »

```
unsigned non_temps_constant (unsigned x, unsigned y, bool secret)
{ if (secret) return y; else return x; }
```

```
unsigned temps_constant (unsigned x, unsigned y, bool secret)
{ return x ^ ((y ^ x) & (-(unsigned)secret)); }
```

# CT-CompCert, un compilateur sécurisé contre les attaques par canaux cachés temporels

Discipline de programmation « temps constant »

```
unsigned temps_constant (unsigned x, unsigned y, bool secret)
{ return x ^ ((y ^ x) & (-(unsigned)secret)); }
```

Version légèrement modifiée de CompCert qui préserve de plus la politique de temps constant au sens de la cryptographie

```
Theorem compilateur-correct:
  forall S C b,
  compile S = OK C →
  exec_CompCertC S b →
  exec_ASM C b.
```

```
Theorem compilateur-préserve-temps-constant:
  forall S C,
  compile S = OK C →
  est-temps—constant S →
  est-temps—constant C.
```

Défis : réutilisation des scripts de preuves de CompCert,
passage à l'échelle des techniques de preuve par simulation

[Barthe, Blazy, Grégoire, Hutin, Laporte, Pichardie, Trieu, POPL'20]

# Conclusion

La vérification déductive apporte des garanties rigoureuses sur l'absence d'erreur dans les logiciels.

CompCert, un jalon de la compilation vérifiée,
une infrastructure disponible pour la communauté de recherche

- **compilation** : ProbCompCert (Boston College, USA), L2C (Tsinghua, Chine), Velus (DIENS), VeriCert (Imperial College, GB), FM-JIT (IRISA), CompCertSSA (IRISA), CompCertO (Yale, USA), CompCert-KVX (Verimag),
- **analyse statique** : Verasco (Inria Paris et Rennes)
- **logiques de programmes** : VST (Princeton, USA), Gillian (Imperial College, GB), VeriFast (KUL, Belgique)

# Questions ?