Spécification et vérification formelles avec l'assistant à la preuve Coq

Xavier Leroy

Inria Paris-Rocquencourt

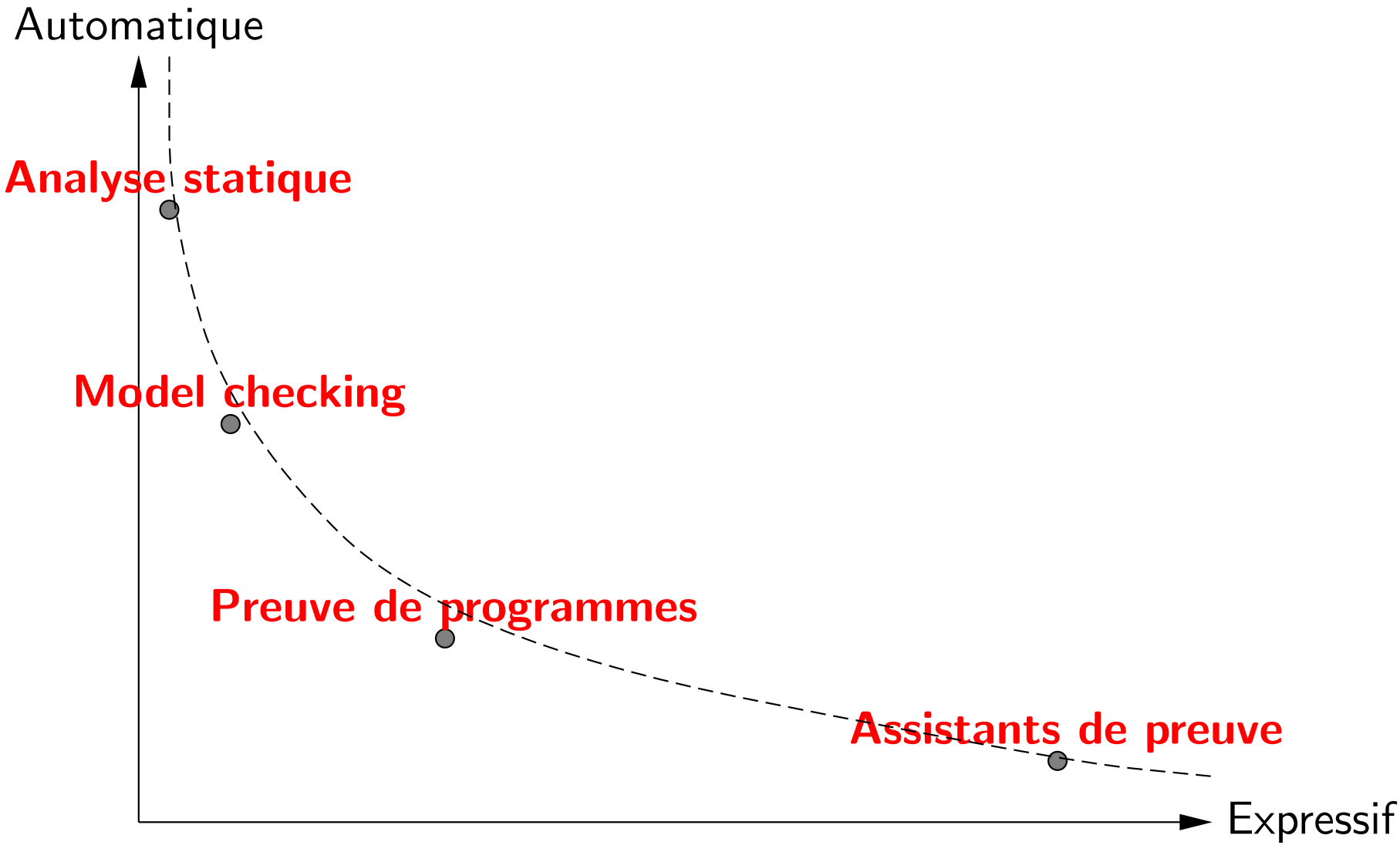
Forum Méthodes Formelles, 2014-02-04

Coq en deux mots

L'assistant à la preuve Coq est un outil pour écrire des spécifications et prouver des propriétés mathématiques.

- Un puissant langage de spécification : Gallina.
  ($\approx$ logique mathématique $\cup$ programmation fonctionnelle.)
- Des commandes appelées ≪tactiques≫ pour développer des preuves en interaction avec l'outil.
  (Ce n'est pas de la démonstration automatique.)
- Un vérificateur de preuves, automatique et très sûr.

Petit panorama d'outils de vérification



Utilisations typiques de Coq

dans le contexte des méthodes formelles pour le logiciel critique

Formaliser et vérifier :

1. Des théories mathématiques de base
   (arithmétique entière ou virgule flottante, algèbre, probabilités, . . .)
   ($\rightarrow$ étude de cas 1 : circuits arithmétiques)

2. Des algorithmes ou protocoles complexes.
   ($\rightarrow$ étude de cas 2 : l'algorithme MJRTY)

3. Certains programmes, à condition qu'ils soient écrits dans le langage fonctionnel pur intégré à Coq.
   ($\rightarrow$ étude de cas 3 : compilation vérifiée)

Les limites de Coq

Coq est inadapté à la vérification entièrement automatique.

(Utilisez plutôt prouveurs SMT, model checking, SAT, . . .)

Coq est peu pratique pour la vérification déductive de programmes écrits dans des langages classiques (C, Java, . . .).

(Utilisez plutôt des prouveurs de programmes comme Frama-C/WP.)

Plan

1. Le langage de spécification Gallina
2. Étude de cas 1 : circuits arithmétiques
3. La preuve en Coq
4. Étude de cas 2 : l'algorithme MJRTY
5. Étude de cas 3 : compilation des expressions arithmétiques
6. Passage à l'échelle : le compilateur C vérifié CompCert
7. Utilisabilité industrielle de CompCert

Gallina, 1 : calculs et fonctions

Proche d'un langage fonctionnel typé comme Caml ou Haskell.

```
Definition sept := 2 * 3 + 1.

Definition moyenne (a b: Z) := (a + b) / 2.

Compute (moyenne sept 20).
```

(réponse : `= 13 : N`)

Gallina, 1 : calculs et fonctions

Fonctions récursives définies par analyse de cas :

```
Fixpoint factorielle (n: nat) :=
  match n with
  | O   => 1
  | S p => n * factorielle p
  end.
```

```
Fixpoint concat (A: Type) (l1 l2: list A) :=
  match l1 with
  | nil => l2
  | h :: t => h :: concat t l2
  end.
```

Note : toutes les fonctions doivent terminer.

Gallina, 2 : logique mathématique

Propositions logiques exprimées avec les connecteurs et quantificateurs habituels :

`->`	implication «si. . . alors»
`/\`	conjonction «et»
`\/`	disjonction «ou»
`~`	négation «non»
`forall`	quantification universelle «pour tout»
`exists`	quantification existentielle «il existe»

Gallina, 2 : logique mathématique

Permet de définir des propriétés logiques et d'énoncer des théorèmes.

```
Definition divise (a b: N) := exists n: N, b = n * a.

Theorem diviseurs_factorielle:
  forall n i, 1 <= i <= n -> divise i (factorielle n).

Definition premier (p: N) :=
  p > 1 /\ (forall d, divise d p -> d = 1 \/ d = p).

Theorem Euclide:
  forall n, exists p, premier p /\ p >= n.
```

Gallina, 3 : types de données inductifs

Définition de types de données par cas :

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.
```

≪Un `nat` est soit `O` soit `S` *n* où *n* est un autre `nat`.≫

```
Inductive list: Type -> Type :=
| nil: forall A, list A
| cons: forall A, A -> list A -> list A.
```

≪Une liste de `A` est soit `nil` soit `cons` d'un `A` et d'une liste de `A`.≫

Gallina, 3 : prédicats inductifs

Le même mécanisme permet de définir des propriétés logiques par une liste de cas :

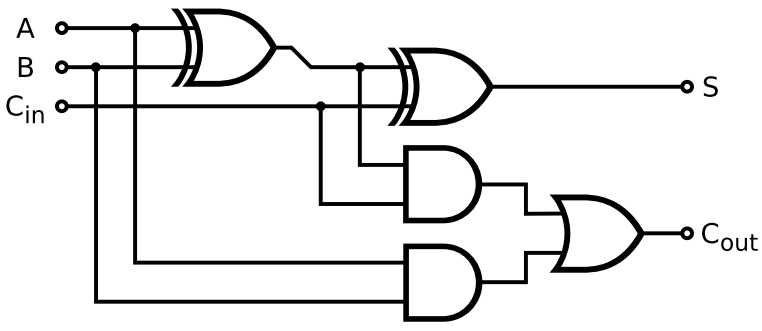
```
Inductive pair: nat -> Prop :=
  | pair_zero:
      pair O
  | pair_plus_2:
      forall n, pair n -> pair (S (S n)).
```

≪Un nombre est pair ssi il est égal à `O` ou il est de la forme `S(S n))` avec `n` pair.≫

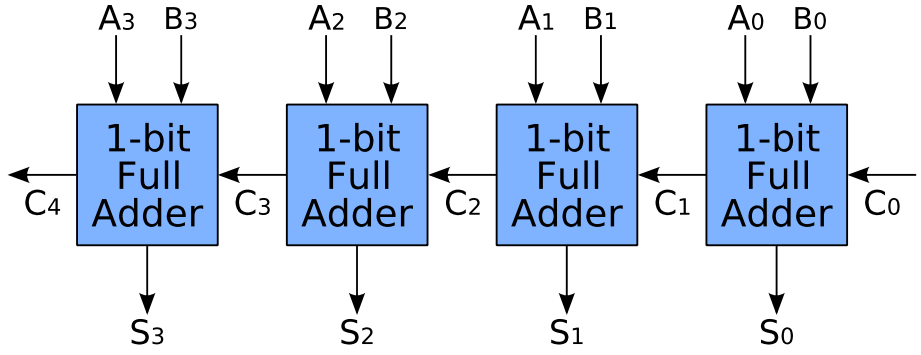
Plan

Les circuits additionneurs

Full adder



4-bit ripple carry adder



Est-ce que ces circuits calculent vraiment l'opération $+$?

$\rightarrow$ Développement Coq `Adders`.

Plan

La preuve en Coq

Un processus interactif, guidé par l'utilisateur.

À tout instant, un ou plusieurs buts qui restent à prouver.

Certaines tactiques résolvent automatiquement des buts simples, p.ex. `eauto` (chaînage arrière), `omega` (arithmétique linéaire), `congruence` (raisonnement équationnel).

D'autres tactiques remplacent le but courant par des sous-buts plus simples, p.ex. `split` (casse une conjonction en deux) ou `induction` (case de base & cas récursifs).

Les tactiques

Environ 20 tactiques essentielles, plus environ 60 tactiques plus spécialisées ou rarement utiles.

Plus : un petit langage de script (`Ltac`) pour définir des combinaisons de tactiques.

Apprentissage assez long (3 mois).

Feedback constant de l'outil : on ne travaille jamais ≪en aveugle≫.

Bonnes ressources pédagogiques : livres, exercices interactifs, vidéo.

Confiance en les preuves

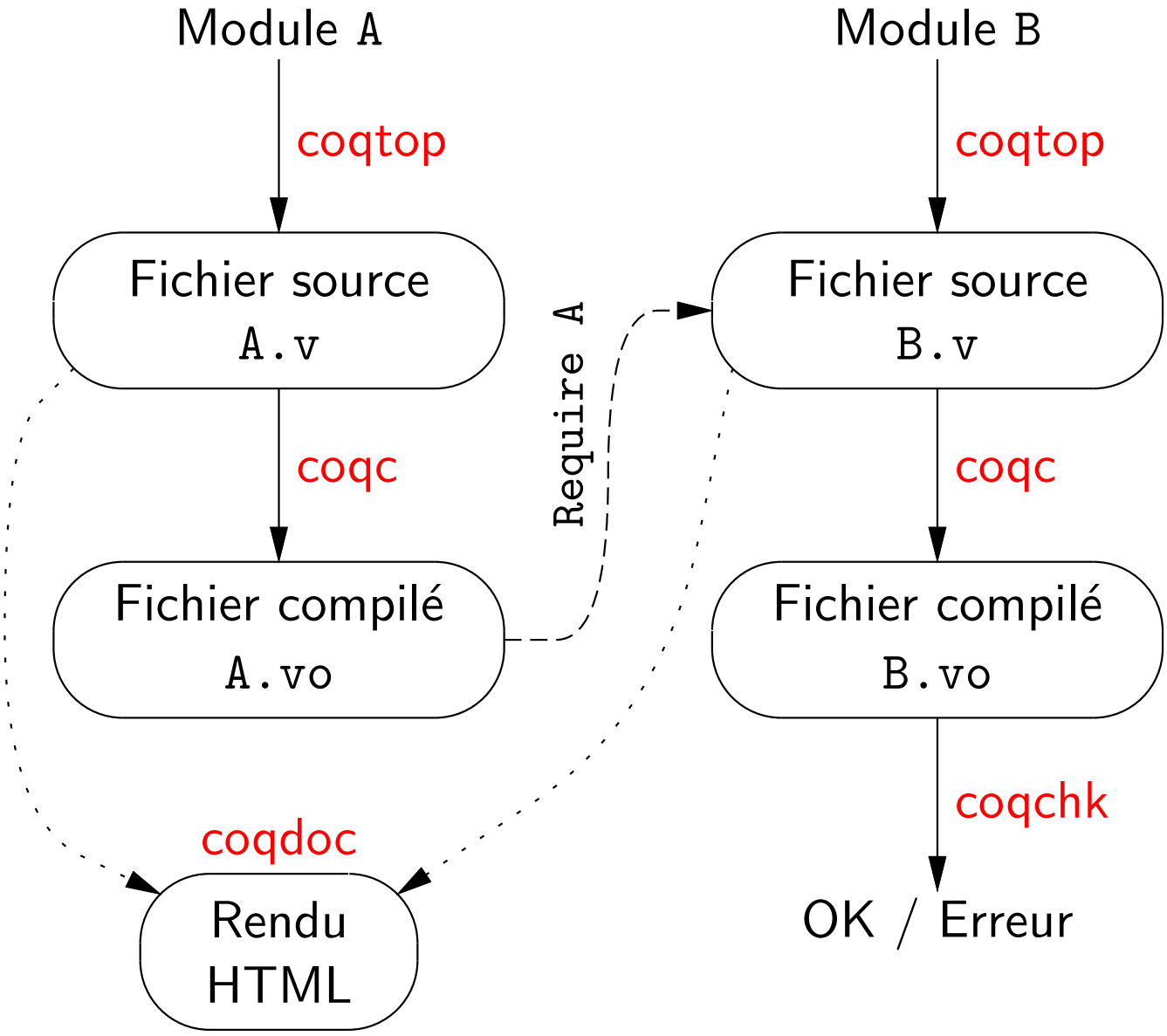
Plusieurs niveaux de vérification :

1- Les tactiques vérifient que leurs conditions d'application sont remplies, et font une erreur sinon.

2- Derrière le rideau, les tactiques construisent incrémentalement un terme de preuve (représentation de bas niveau, très précise).
Ce terme de preuve est vérifié par le noyau de Coq :

- À la fin de chaque preuve interactive (le `Qed.`).
- Lorsqu'on compile le développement en batch avec `coqc`.
- Optionellement, avec l'outil `coqchk`.

Workflow Coq



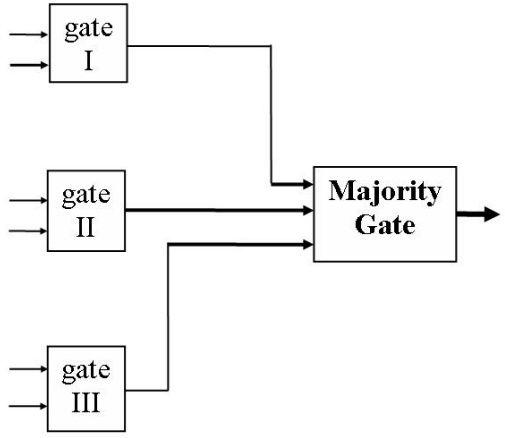
Développement interactif

Compilation

Revérification

Plan

Dépouillement efficace d'un vote



N électeurs votent parmi M candidats. Déterminer si un candidat obtient la majorité absolue, et si oui, lequel.

N calculateurs redondants produisent N valeurs parmi M. Déterminer si une valeur est produite plus de $N/2$ fois, et si oui, laquelle.

Challenge : trouver un algorithme efficace, en temps linéaire $O(N)$ et en espace constant.

Cas simple : N petit

Exemple : $N = 3$, redondance modulaire triple.

si $vote[0] = vote[1]$, le gagnant est $vote[0]$
si $vote[1] = vote[2]$, le gagnant est $vote[1]$
si $vote[2] = vote[0]$, le gagnant est $vote[2]$
sinon, pas de gagnant

Autre cas simple : M petit

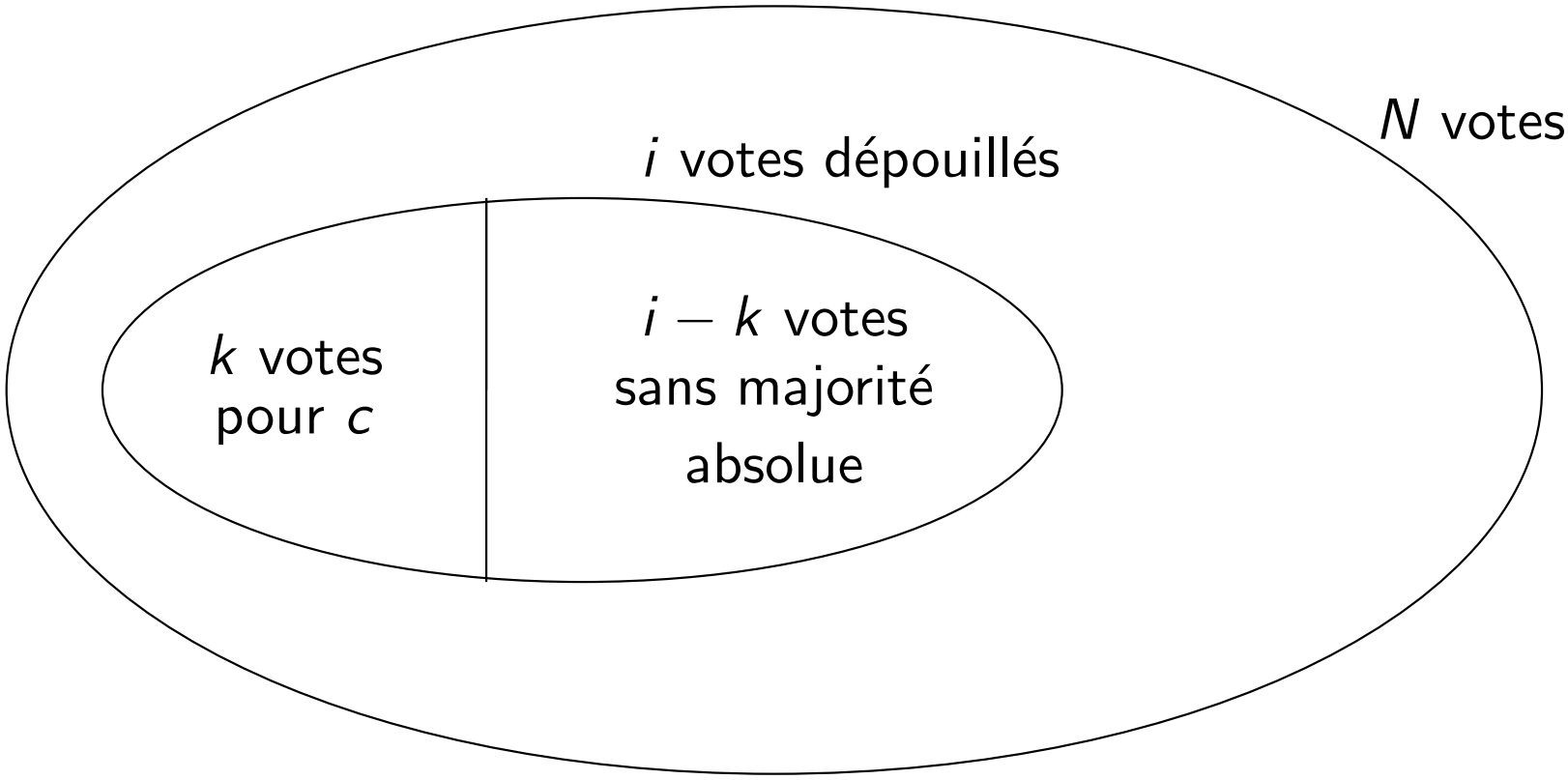
Exemple : $M = 2$, deux valeurs/candidats, «0» et «1».

$n_0 := 0$
$n_1 := 0$
pour $i = 0$ à $N - 1$:
    si $vote[i] = 0$ incrémenter n_0
    si $vote[i] = 1$ incrémenter n_1
fin
si $n_1 > n_0$, le gagnant est «1»
si $n_0 > n_1$, le gagnant est «0»
sinon, pas de gagnant.

L'algorithme MJRTY (Boyer & Moore, 1980)

$c :=$ un candidat quelconque
$k := 0$
pour $i = 0$ à $N - 1$:
    si $k > 0$:
        si $vote[i] = c$ alors $k := k + 1$ sinon $k := k - 1$
    sinon :
        $c := vote[i]$; $k := 1$
fin
compter le nombre n de votes pour le candidat c
si $n > N/2$, le gagnant est c
sinon, pas de gagnant

Pourquoi ça marche ?



À la fin de la boucle ($i = N$), on ne sait pas si le candidat c a la majorité absolue, mais on sait qu'aucun autre candidat ne l'a.

Vérification formelle

→ Développement Coq `Majority.v`

Plan

La notation polonaise inverse

Notation algébrique : $2 \times (3 + 4)$

$\downarrow$

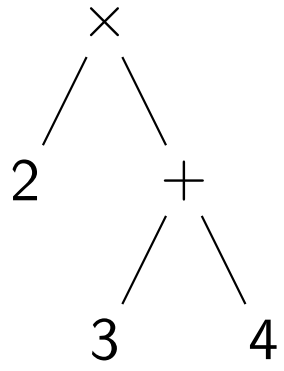
Notation polonaise : $2 \ 3 \ 4 \ + \ \times$

Comment passer de la première à la seconde ?

Pourquoi les deux notations calculent le même résultat ?

Langage source : expressions arithmétiques

Représentées par des arbres de syntaxe abstraite.




```
Inductive expr : Type :=
  | Const (n: Z)
  | Sum (e1 e2: expr)
  | Diff (e1 e2: expr)
  | Prod (e1 e2: expr).
```

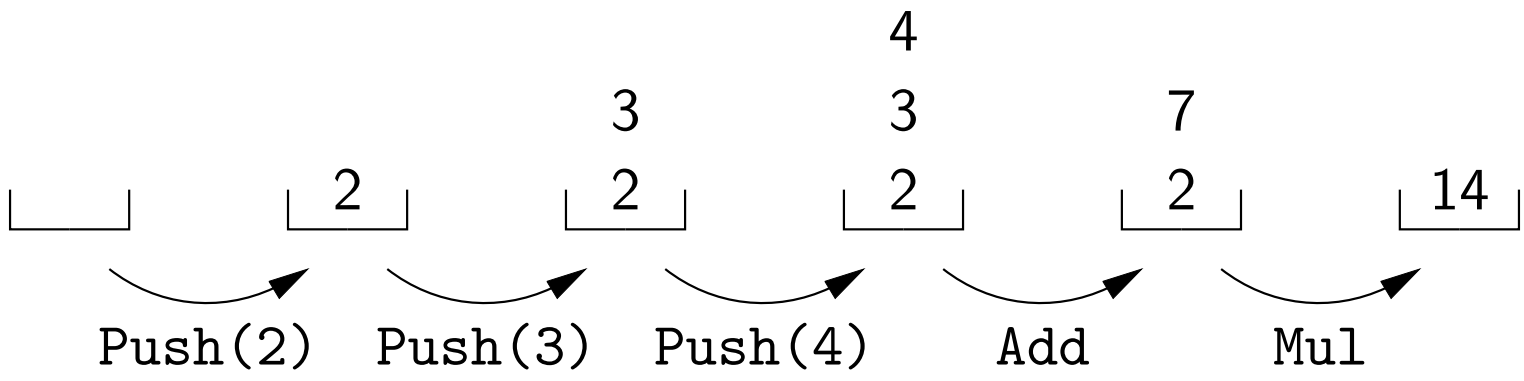
La sémantique d'une expression est la valeur entière qu'elle dénote, p.ex. 14 dans l'exemple ci-dessus.

Machine cible : calculateur à pile

Le jeu d'instructions :

`Push(`n`)`	empile la constante n
`Neg`	change le signe du sommet de la pile
`Add`	dépile deux nombres, empile leur somme
`Mul`	dépile deux nombres, empile leur produit

Sémantique : par transitions successives, chaque instruction faisant évoluer l'état de la machine, c.à.d. la pile.



Le schéma de compilation

Traduction expression $\rightarrow$ séquence d'instructions.

Cas de base : l'expression est une constante n

- Produire l'instruction `Push`(n).

Cas récursif : l'expression est de la forme $e_1 + e_2$

- Produire récursivement le code pour e_1
  (sommet de pile : la valeur de e_1)
- Produire récursivement le code pour e_2
  (sommet de pile : valeur de e_2 ; en dessous : valeur de e_1)
- Produire l'instruction `Add`
  (sommet de pile : valeur de e_1 + valeur de e_2)

Correction du compilateur

Une propriété de préservation sémantique :

Théorème
Pour toute expression e, l'exécution du code machine produit pour e se termine sans erreur avec la valeur de e au sommet de la pile.

→ Développement Coq `RPN.v`

Plan

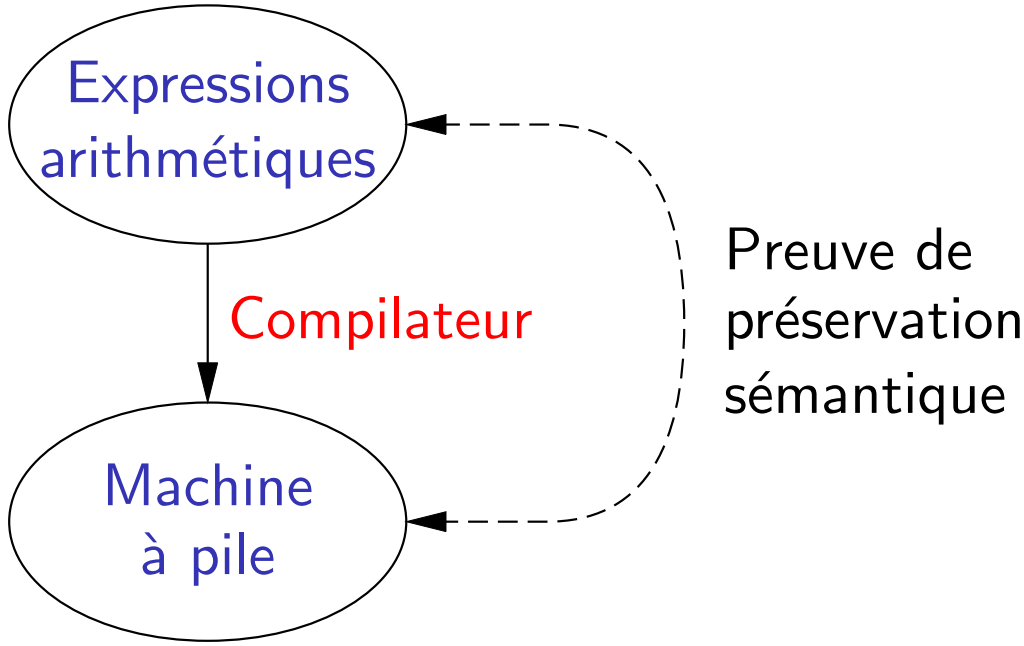
Le projet CompCert

X. Leroy, S. Blazy, et al ; depuis 2004

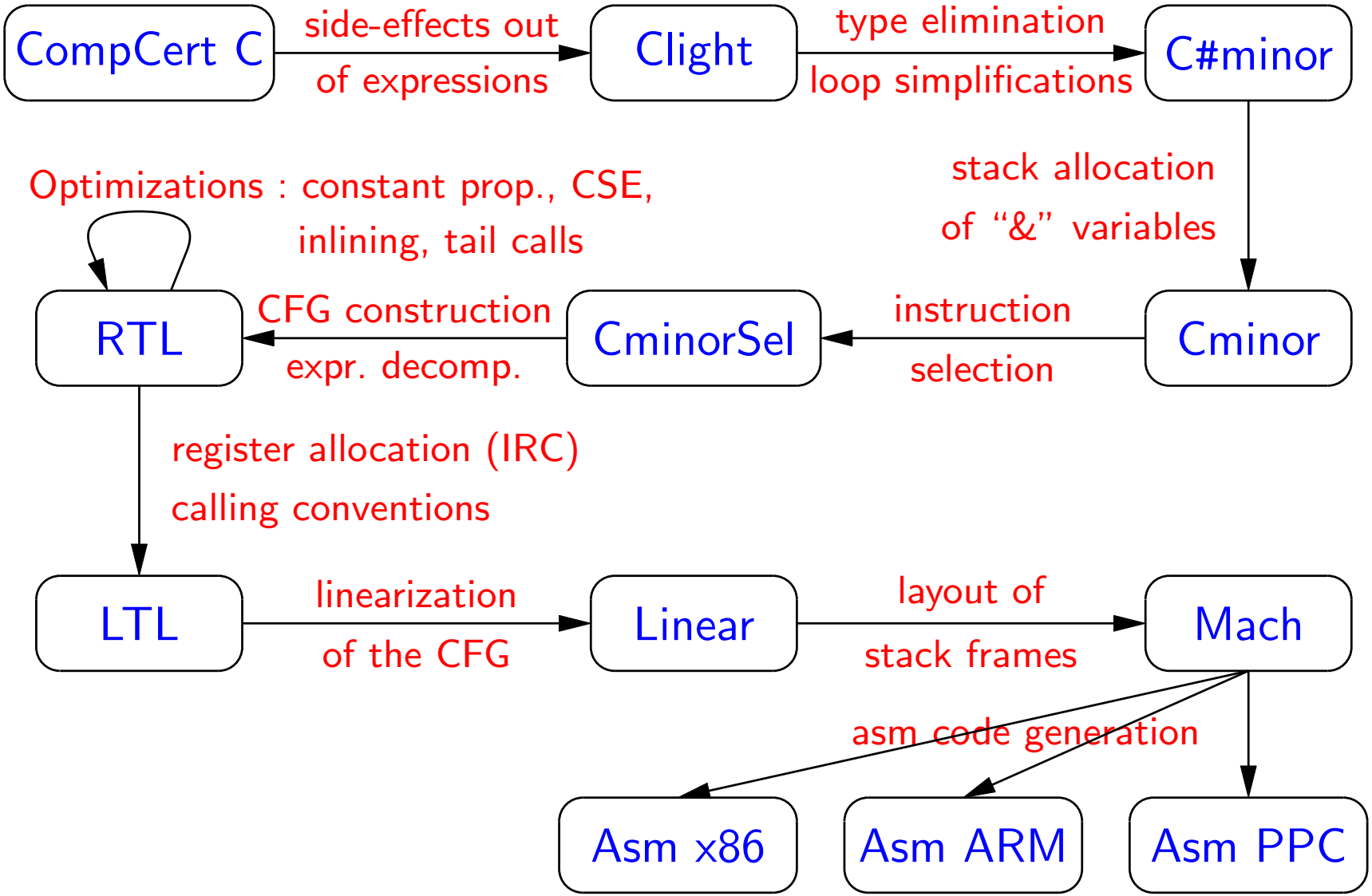
Développer et prouver correct un compilateur réaliste, utilisable pour le logiciel embarqué critique.

- Langage source : presque tout ISO C 99.
  (Sauf : `long double`, variable-length arrays, `switch` non structuré)
- Langages cible : assembleurs PowerPC, ARM, x86 32 bits.
- Des optimisations pour produire du code assez efficace
  (2 fois plus rapide que `gcc -O0` ; 10% plus lent que `gcc -O1`).
- Des preuves de préservation sémantique pour la plupart des passes du compilateur.

De notre 3ième étude de cas...



. . . à la partie vérifiée de CompCert

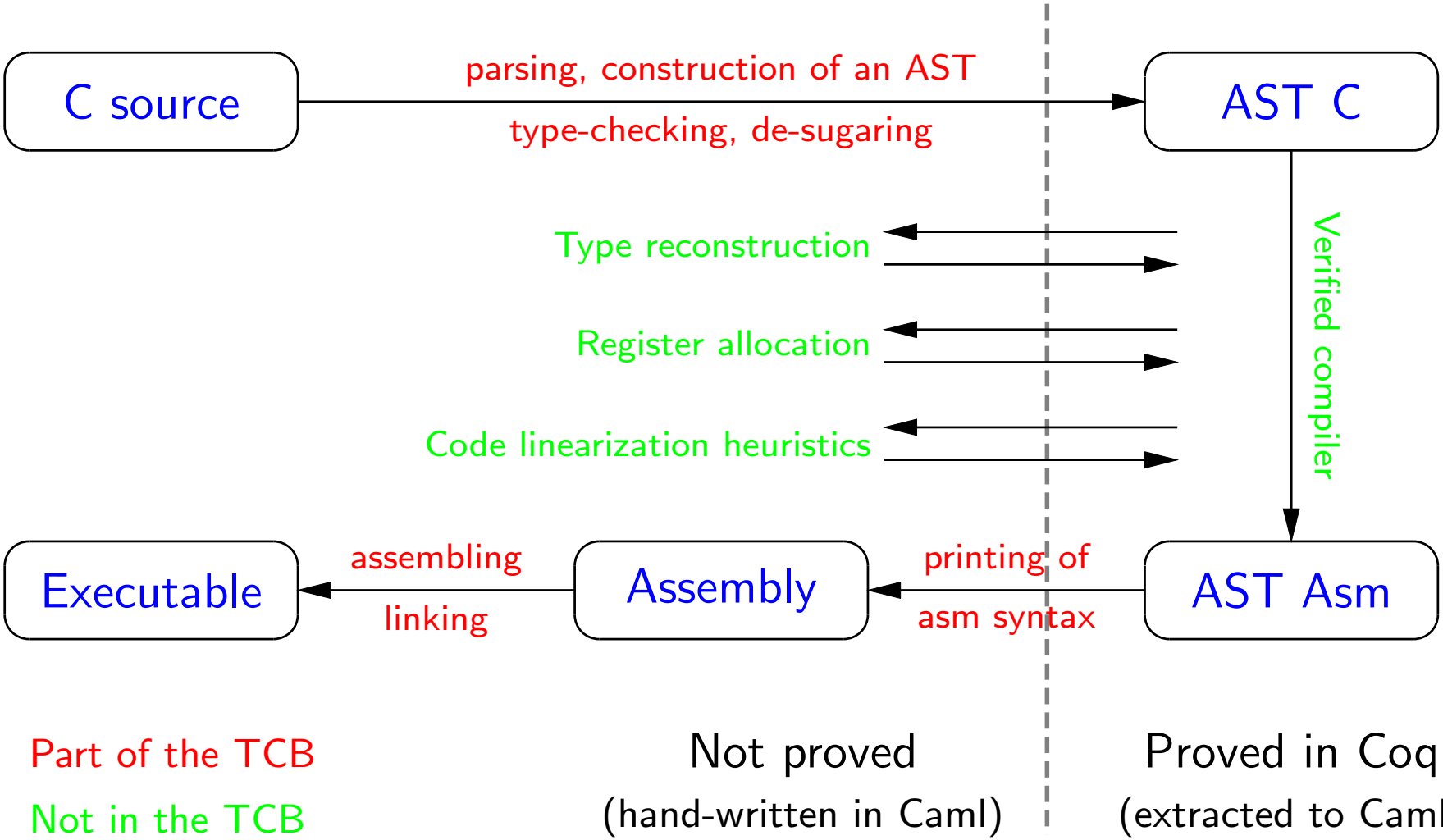


Utilisations de Coq

CompCert utilise Coq de manière essentielle :

- Pour spécifier la syntaxe et la sémantique des langages source, cibles, et intermédiaires.
- Pour programmer la plupart des passes de compilation. (Exécutabilité via l'extraction Coq $\rightarrow$ OCaml.)
- Pour mener à bien les preuves de préservation sémantique.

Le compilateur CompCert C au complet



Passage à l'échelle

	Étude de cas 3	CompCert
Spécifications des langages	40 L	7 000 L
Code du compilateur	8 L	10 000 L
Preuves de correction	40 L	50 000 L
Bibliothèques	–	10 000 L
Effort total	2 h	5 ans

Plan

Utilisabilité de CompCert dans l'industrie

$\rightarrow$ Exposé de Jean Souyris, Airbus.

Référence :

Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, et Jean Souyris.
Formally verified optimizing compilation in ACG-based flight control software.
Embedded Real Time Software and Systems (ERTS 2012).