Comment faire confiance à un compilateur?

Xavier Leroy

INRIA Paris-Rocquencourt

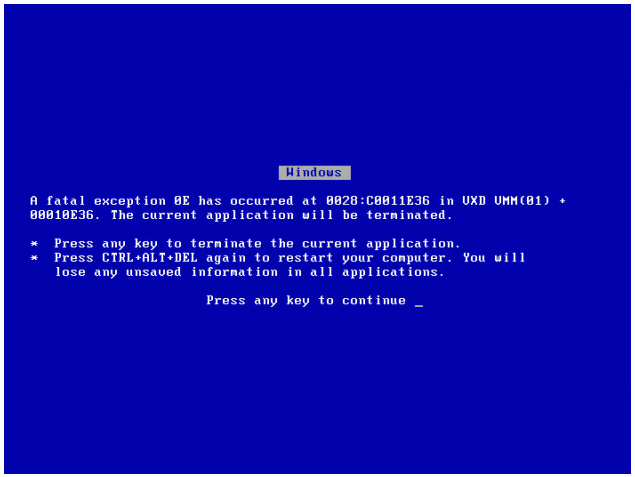
Colloquium J. Morgenstern, 2009-10-08

Contexte : les méthodes formelles et la vérification de programmes

Le bug

«En informatique, il y a le bug ; c'est comme ça, vous n'y pouvez rien.»

(Gérard Berry, Collège de France, 17/01/2008.)



Tolérable ?

Intolérable !

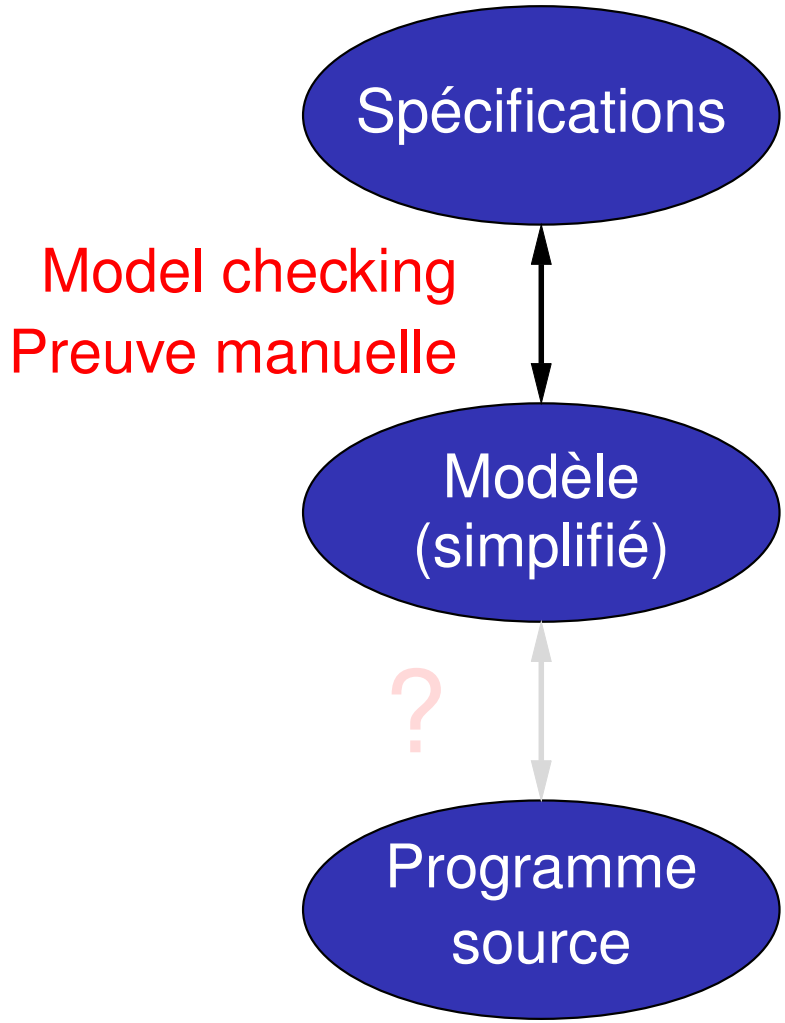
Le logiciel critique

Les méthodes classiques :

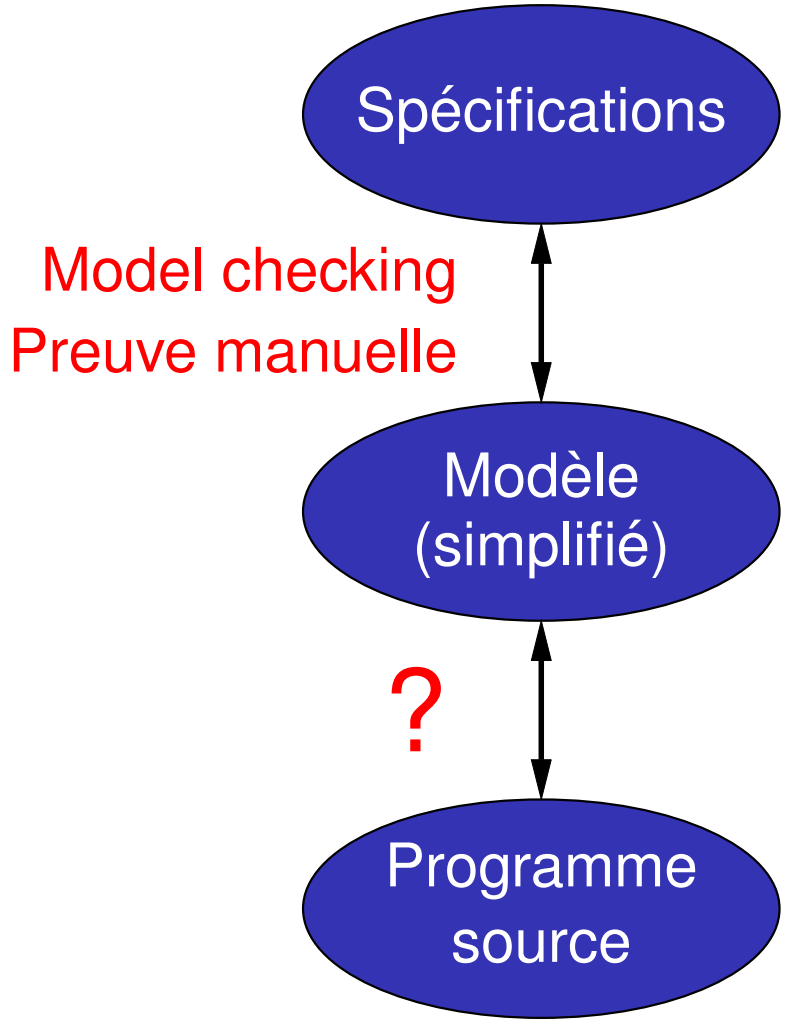
- Méthodologie de développement extrêmement rigoureuse.
- Des tests, des tests, et encore des tests.

Lorsque tout cela ne suffit plus ou coûte trop cher :
les méthodes formelles (approches mathématiques)

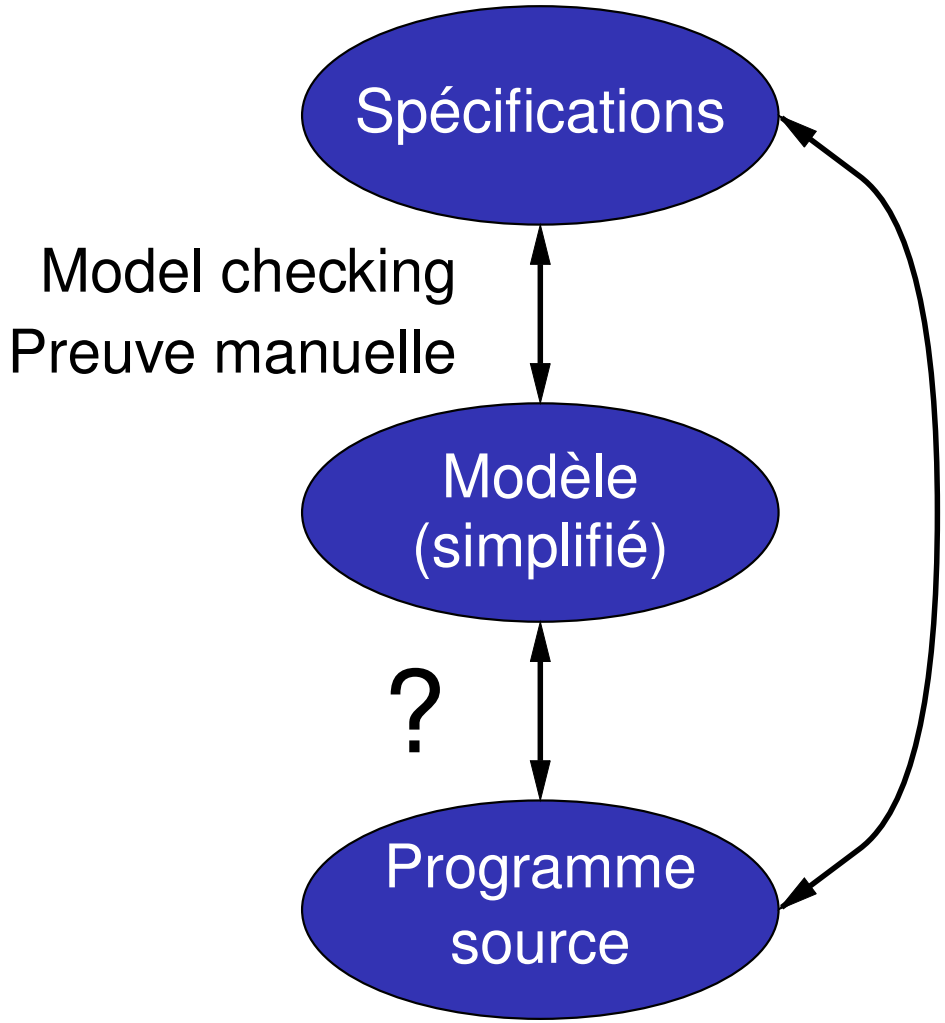
Vérification sur un modèle



Vérification sur un modèle



Vérification formelle de programmes



Analyse statique

Preuve de programme

L'analyse statique pour la sûreté

Établir automatiquement des propriétés de sûreté de base.

Signaler les erreurs potentielles (*debugging statique*).

Exemple : sûreté de la mémoire en C.

- Accès aux tableaux dans les bornes.
  (Pas de *buffer overrun*.)
- Pas d'accès au pointeur nul.
- Pas d'accès après un `free` ; pas de double `free`.

Exemple d'analyse statique

```
double dotproduct(int n, double * a, double * b)
{
    ...
}

int main()
{
    double * a, * b, d;
    a = calloc(na, sizeof(double));
    b = calloc(nb, sizeof(double));
    /* Remplir a et b */
    d = dotproduct(n, a, b);
    ...
}
```

L'analyseur statique dit OK s'il peut prouver `n <= na` et `n <= nb`.
Il signale un bug potentiel sinon.

Exemple d'analyse statique

Propriétés de l'arithmétique de la machine :

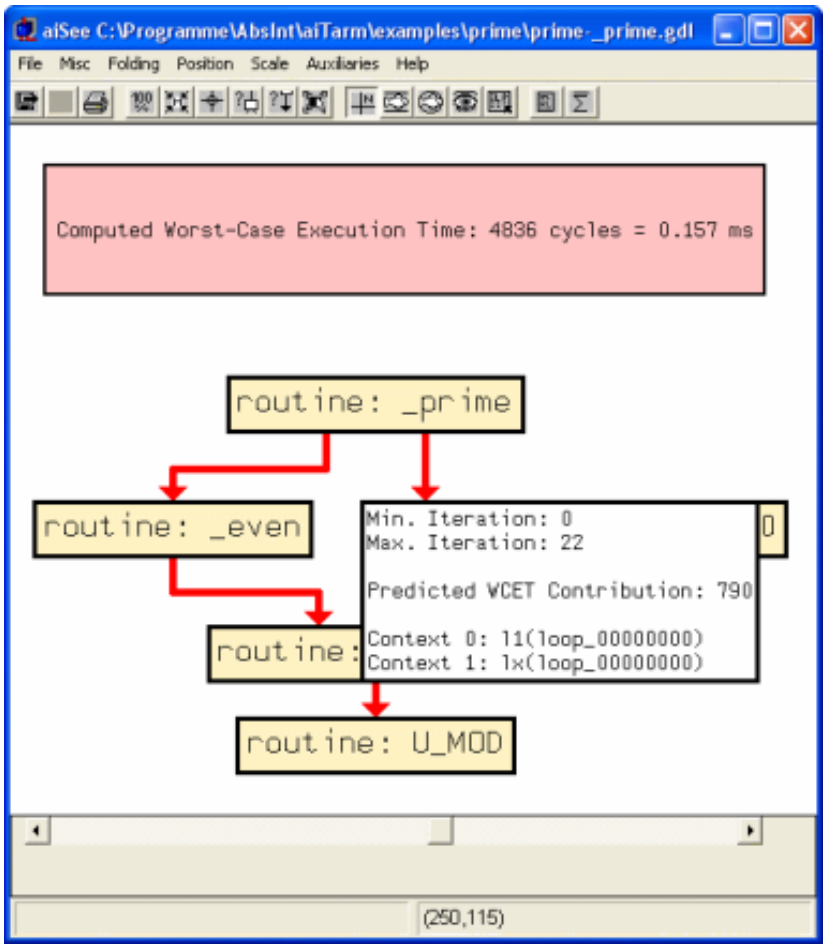
- Pas de débordements dans les calculs entiers.
- Pas de débordements (`Inf`, `NaN`) dans les calculs flottants.
- Borner les imprécisions des calculs flottants.

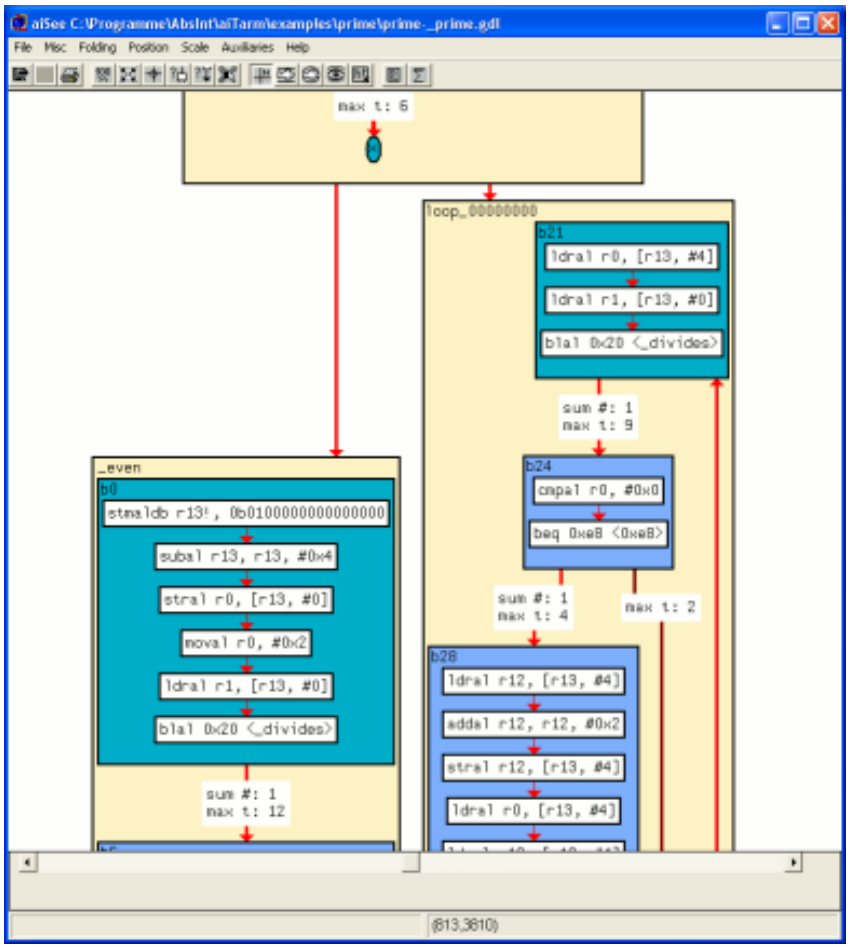
Astrée (ENS) :
si `a[i]` $\in [0.0, 1.5]$ et `b[i]` $\in [0.0, 1.5]$,
alors `dotproduct`$(10, \texttt{a}, \texttt{b}) \in [0.0,\ 22.5000007426]$.

Fluctuat (CEA) :
si les `a[i]` et `b[i]` sont exacts à ϵ près, alors
`dotproduct`$(10, \texttt{a}, \texttt{b})$ est exact à $P(\epsilon)$ près.

Exemple d'analyse statique

Bornes fines sur le temps d'exécution (Absint, Ait WCET).





Preuve de programmes

Le programmeur annote le source avec des formules logiques : préconditions et postconditions de fonctions, invariants de boucles, ...

L'outil engendre des obligations de preuves qui seront démontrées soit automatiquement, soit interactivement.

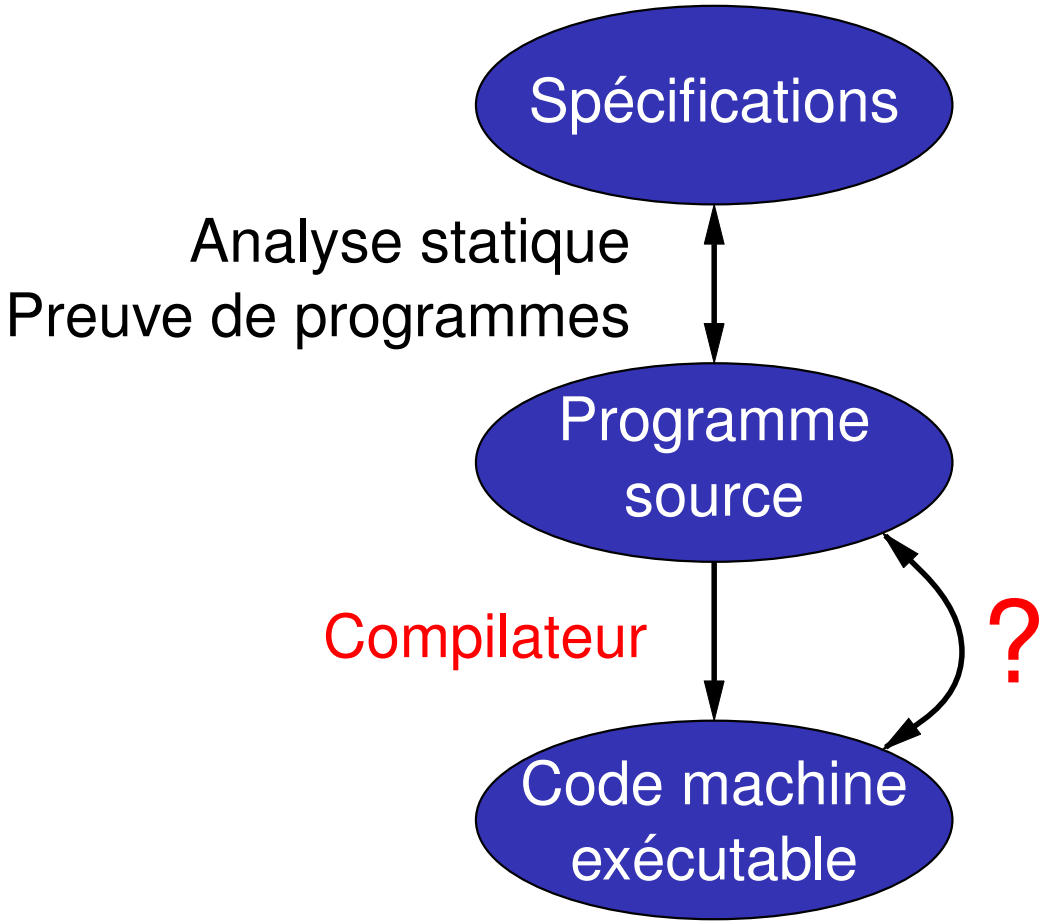
Exemple de preuve de programmes (Frama-C)

```
/*@ predicate is_min(int t[],int n,int min) {
       (\forall int i ; 0 <= i < n => min <= t[i]) &&
       (\exists int i ; 0 <= i < n && min == t[i])
    }
  @*/

/*@ requires n > 0 && \valid_range(t,0,n)
  @ ensures is_min(t,n,\result)
  @*/
int min(int t[],int n) {
  int i ;
  int tmp = t[0] ;
  /*@ invariant 1 <= i <= n && is_min(t,i,tmp) @*/
  for (i=1 ; i < n ; i++) {
     if (t[i] < tmp) tmp = t[i] ;
  }
  return tmp ;
}
```

Les compilateurs et la confiance que l'on peut leur accorder

Le problème du compilateur



Ce qui est formellement vérifié est le programme source, pas le code exécutable. Est-on sûr que le compilateur préserve les garanties ainsi obtenues ?

Que fait un compilateur ?

Au sens large : toute traduction automatique d'un langage informatique vers un autre.

Au sens restreint : traduction efficace ("optimisante") d'un langage source (compréhensible par les programmeurs) vers un langage machine (compréhensible par le *hardware*).

Un domaine mature :

- Plus de 50 ans. . . (Fortran I : 1957)
- Énorme corpus d'algorithmes (optimisations).
- Nombreux compilateurs qui effectuent des transformations très subtiles.

Un exemple de compilation optimisante

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compilé pour le processeur Alpha au niveau d'optimisation maximal, puis décompilé manuellement en C...

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17 : a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19 : dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5 :  return dp;
L14 : a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17 : a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19 : dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5 :  return dp;
L14 : a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17 : a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19 : dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5 :  return dp;
L14 : a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

Les bugs dans les compilateurs

Comme tout programme complexe, les compilateurs contiennent des bugs.

La plupart font juste “planter” la compilation (pas très grave).

Certains font produire silencieusement un exécutable faux à partir d’un source correct (très gênant).

Un exemple de bug de compilateur

```
      subroutine bug1(expnt)
      implicit none
      double precision zeta
      common /bug1_area/zeta(3)
      double precision expnt(3)
      integer k, kkzc
      kkzc=0
      do k=1,3
         kkzc = kkzc + 1
         zeta(kkzc) = expnt(k)
      enddo
c     the following line activates the bug
      call bug1_activator(kkzc)
      end
```

Est censé copier le tableau `expnt` dans le tableau `zeta`.
En GCC 3.3.2, la copie est décalée d'une case (bug dans la passe GCSE).

Bug ou pas ?

L'outil CIL d'analyse statique de code C, version 1.3.5, effectue la simplification suivante. Est-elle correcte ?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list ** p)
{
  return ((*p)->tail = NULL);
}
```

```
(*p)->tail = NULL;
return (*p)->tail;
```

Non, pas dans le cas d'une liste circulaire :

Bug ou pas ?

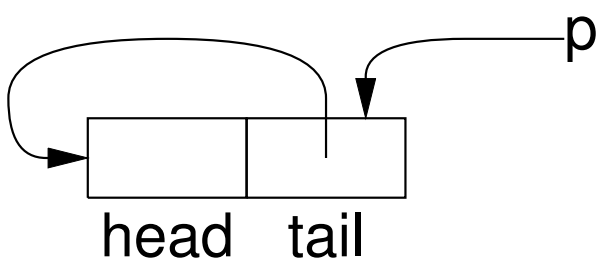
L'outil CIL d'analyse statique de code C, version 1.3.5, effectue la simplification suivante. Est-elle correcte ?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list ** p)
{
  return ((*p)->tail = NULL);
}
```

```
(*p)->tail = NULL;
return (*p)->tail;
```

Non, pas dans le cas d'une liste circulaire :



Bug ou pas ?

GCC 4.x effectue l'optimisation suivante. Est-elle correcte ?

```
x = p->tail;
if (p == NULL) return -1;
...
```

```
x = p-> tail;
/* test supprimé */
...
```

Oui d'après le standard C, mais les développeurs du noyau Linux ne sont pas d'accord.

Bug ou pas ?

GCC 4.x effectue l'optimisation suivante. Est-elle correcte ?

```
x = p->tail;
if (p == NULL) return -1;
...
```

```
x = p-> tail;
/* test supprimé */
...
```

Oui d'après le standard C, mais les développeurs du noyau Linux ne sont pas d'accord.

Vérifier formellement le compilateur

Appliquer les méthodes formelles au compilateur lui-même pour établir un théorème de préservation sémantique :

Théorème

Pour tous les codes source S,
si le compilateur transforme S en le code machine C,
sans signaler d'erreur de compilation,
et si S a une sémantique bien définie,
alors C se comporte comme S.

Corollaire

Sous les hypothèses ci-dessus + le déterminisme du code machine, toute propriété vraie pour le comportement observable de S est également vraie pour celui de C.

Vérifier formellement le compilateur

Appliquer les méthodes formelles au compilateur lui-même pour établir un théorème de préservation sémantique :

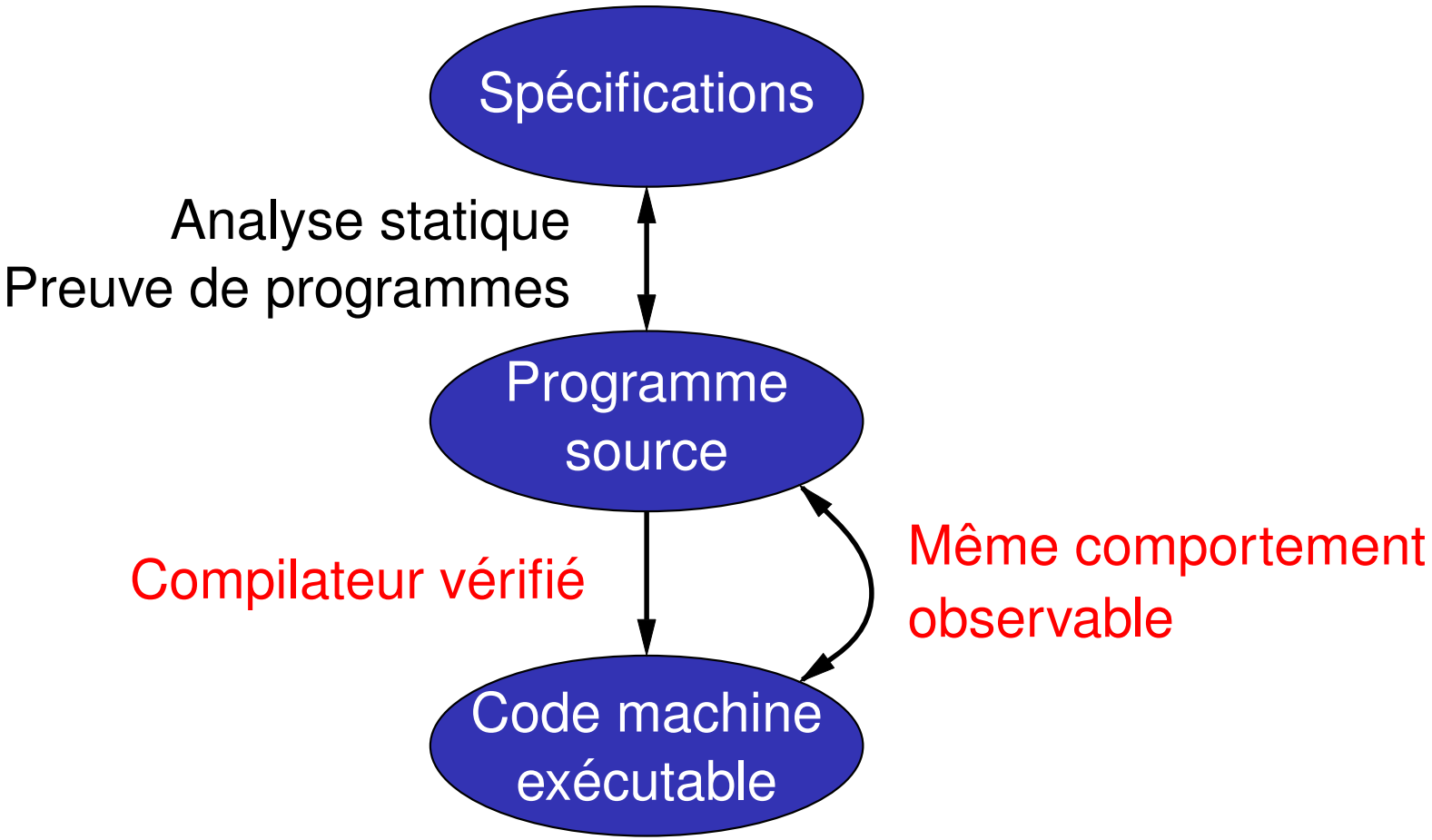
Théorème

Pour tous les codes source S,
si le compilateur transforme S en le code machine C,
sans signaler d'erreur de compilation,
et si S a une sémantique bien définie,
alors C se comporte comme S.

Corollaire

Sous les hypothèses ci-dessus + le déterminisme du code machine, toute propriété vraie pour le comportement observable de S est également vraie pour celui de C.

Utilisation d'un compilateur vérifié



Les garanties obtenues par vérification formelle du source s'étendent automatiquement au code exécutable.

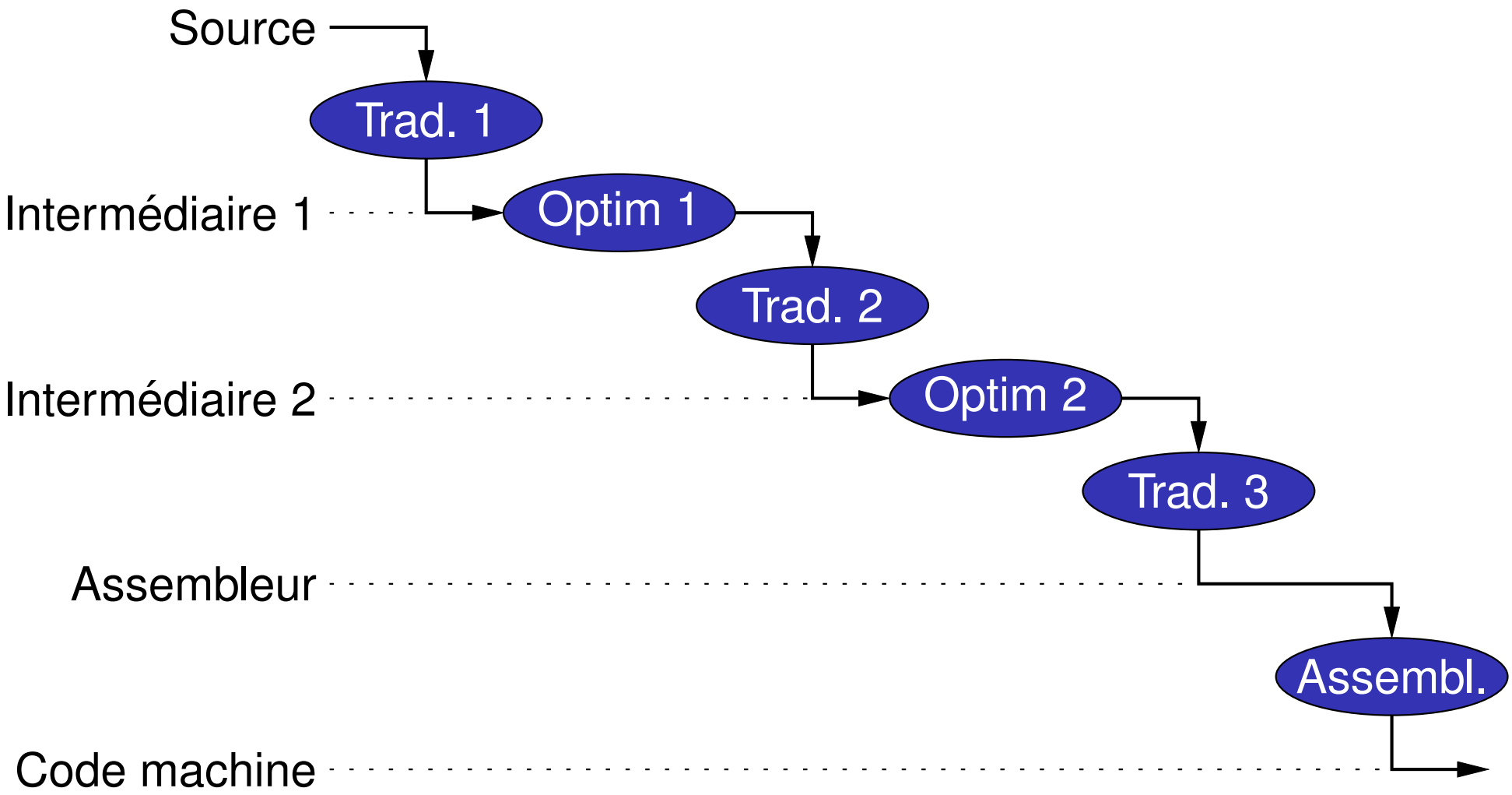
Pourquoi serait-ce faisable ?

Un compilateur est un programme beaucoup plus gros que les fragments de code que l'on sait typiquement prouver corrects.

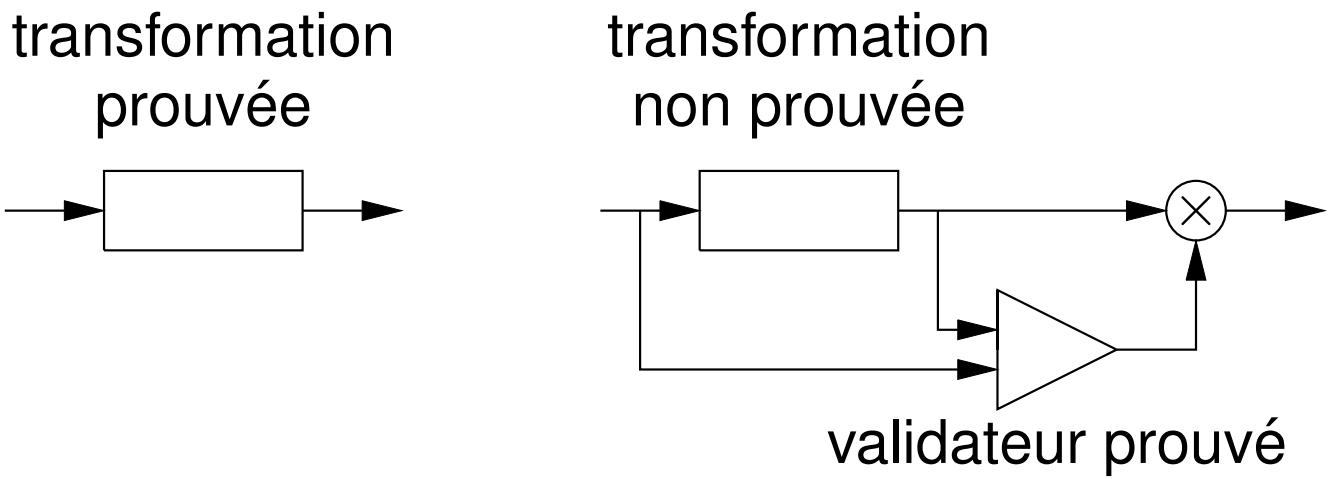
Plusieurs raisons font que la preuve de compilateurs est faisable :

1. La spécification est relativement simple : il "suffit" de se mettre d'accord sur les sémantiques des langages source et cible.
2. Le compilateur n'est pas embarqué et donc peut être écrit dans un langage de haut niveau qui se prête bien à la preuve (langage fonctionnel pur).
3. Le découpage naturel en de multiples passes de compilation
   $\Rightarrow$ modularisation de la preuve.
4. Un compilateur a toujours le droit d'échouer.
   $\Rightarrow$ on peut valider a posteriori des calculs non prouvés.

Découpage en passes



Transformation prouvée vs. validation prouvée



Pour certaines transformations, la preuve du validateur est 10 fois plus petite que celle de la transformation.

Le projet CompCert

Le projet CompCert

(INRIA Gallium, INRIA Marelle, CNAM/ENSIIE, U. Paris-Diderot)

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

- Langage source : un sous-ensemble de C.
- Langage cible : assembleur du processeur PowerPC et ARM.
- Produit du code raisonnablement compact et efficace
  $\Rightarrow$ plusieurs optimisations.

La preuve est faite “sur machine” en utilisant l’assistant de preuve Coq.

Le sous-ensemble de C traité

Traité :

- Types entiers, flottants, tableaux, pointeurs, `struct`, `union`.
- Arithmétique, arithmétique de pointeurs.
- Contrôle : `if`/`then`/`else`, boucles, `switch` simples. `goto`.
- Fonctions, récursion, pointeurs vers fonctions.

Non traité :

- Types `long long` et `long double`.
- `switch` non structurés, `longjmp`/`setjmp`.
- Fonctions à nombre variable d'arguments.
- Passage de `struct` et `union` par valeur.

Diagramme de la partie prouvée du compilateur

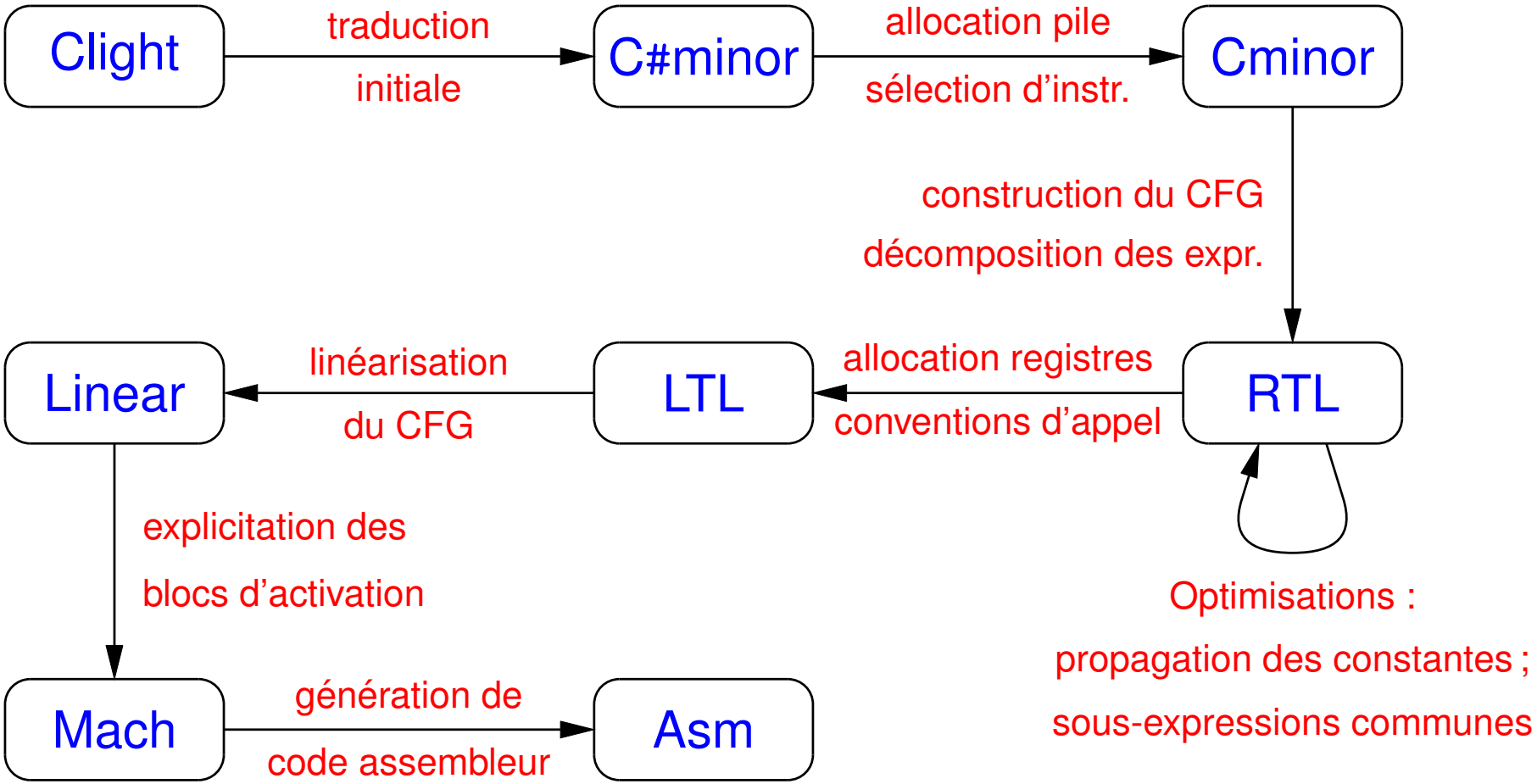
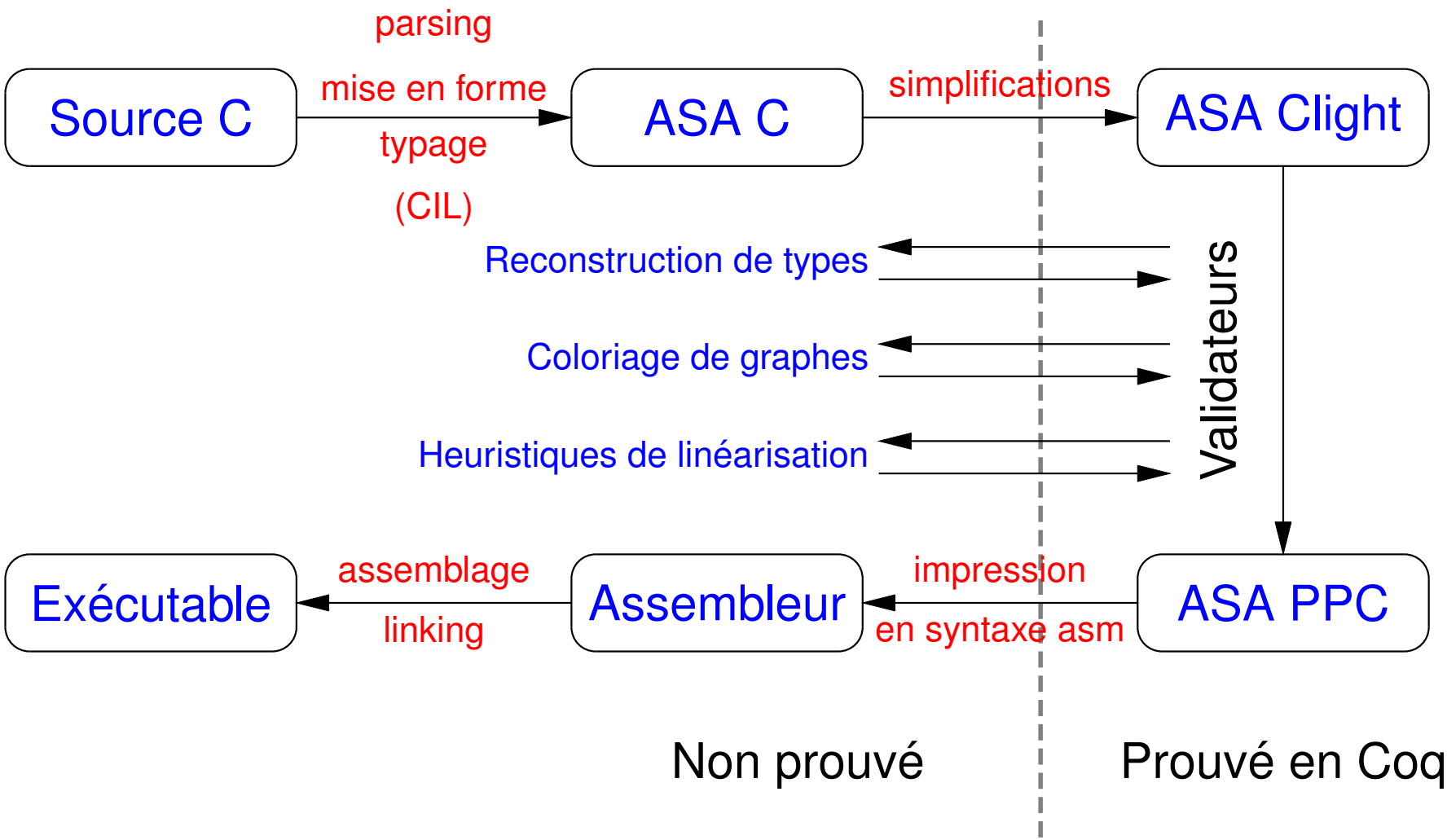


Diagramme du compilateur Compcert entier



Vérifié en Coq

La preuve de correction du compilateur (préservation de la sémantique) est faite entièrement sur machine, avec l'assistant de preuve Coq.

```
Theorem transf_c_program_correct :
    forall prog tprog behavior,
    transf_c_program prog = OK tprog ->
    Clight.exec_program prog behavior ->
    Asm.exec_program tprog behavior.
```

Les comportements observables sont :

- Terminaison + trace finie d'événements d'entrée/sortie.
- Divergence + trace finie ou infinie.

48000 lignes de Coq, 3 hommes-années. (Pour 8000 lignes de code.)

Programmé en Coq

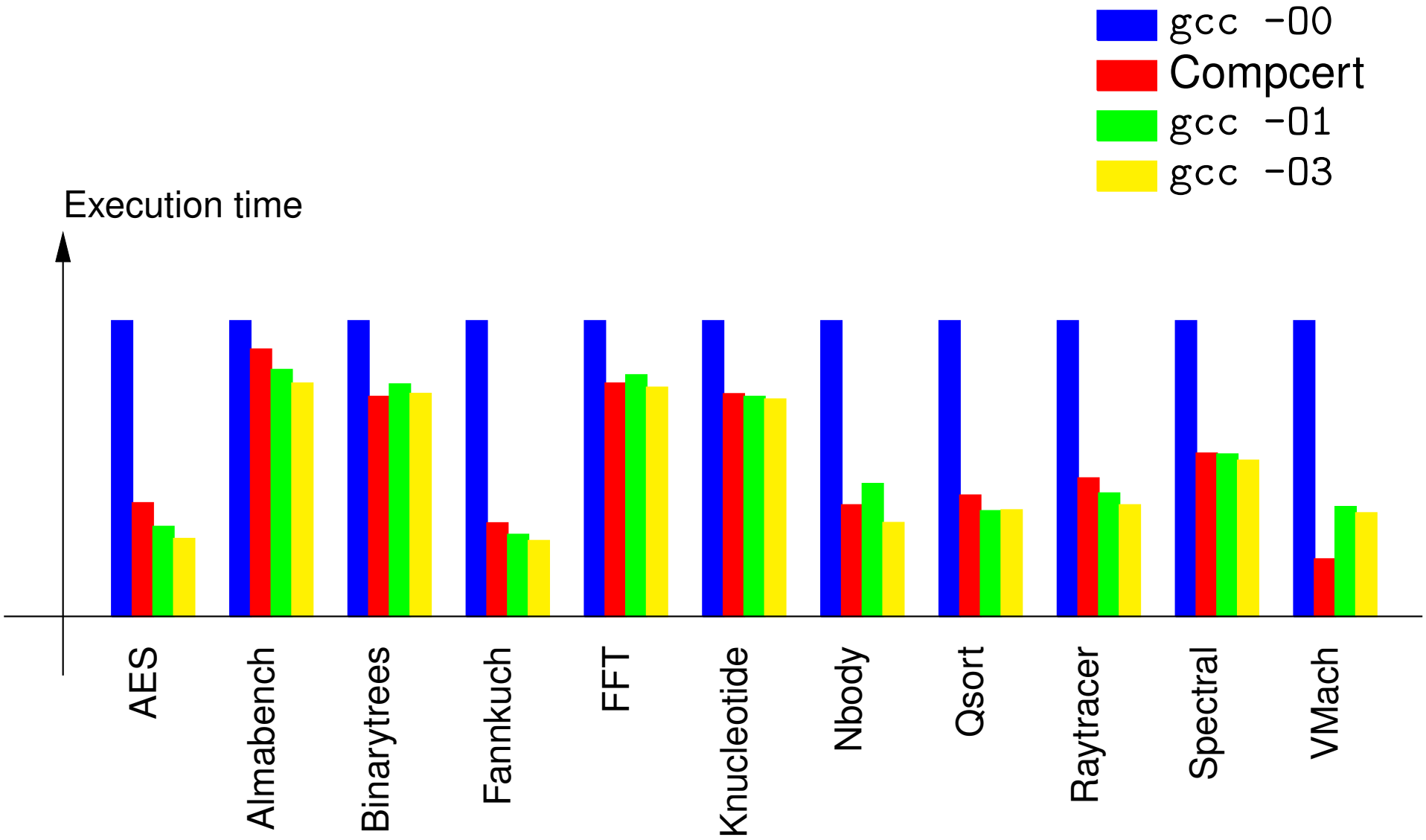
Toutes les parties vérifiées du compilateur sont programmées directement dans le langage de spécification de Coq, en style fonctionnel pur.

- Utilisation de monades pour traiter les erreurs et les états.
- Structures de données purement fonctionnelles (persistantes).

Le mécanisme d'extraction de Coq produit du code Caml exécutable à partir de ces spécifications.

Sans doute le plus gros programme extrait d'un développement Coq.

Performances du code produit



Zoom technique : factorisation de calculs redondants

Factorisation de calculs redondants (CSE)

Redondances explicites dans le code source :

```
return 1.0 / ((i+j) * (i+j+1)/2 + i + 1) ;

int k = i + j ; return 1.0 / (k * (k+1)/2 + i + 1) ;
```

Redondances implicites dans le source, apparaissent après traduction en langage intermédiaire :

```
x = a[i] ; a[i] = a[j] ; a[j] = x ;

--->    t1 = i * 4 ;          t6 = i * 4 ;
        t2 = a + t1 ;         t7 = a + t6 ;
        x = load(t2) ;        store(t7, t5) ;
        t3 = j * 4 ;          t8 = j * 4 ;
        t4 = a + t4 ;         t9 = a + t8 ;
        t5 = load(t4) ;       store(t9, x) ;
```

L'algorithme de *value numbering*

Une forme d'exécution symbolique du code intermédiaire.

État du calcul symbolique :

- Une association ϕ d'inconnues X aux variables du programme.
- Un ensemble E d'équations entre inconnues, p.ex. $\{X_3 = X_1 * 4; X_4 = X_2 + X_3\}$.

Exécution symbolique d'une instruction $t_3 = t_1 + t_2$:

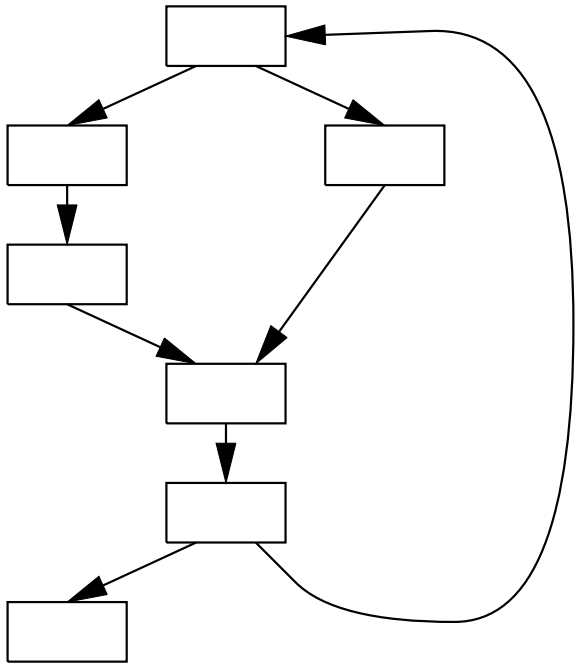
- Trouver les inconnues X_1, X_2 associées aux arguments t_1, t_2.
- Si E contient déjà une équation $X = X_1 + X_2$: associer X au résultat t_3.
- Sinon, allouer une nouvelle inconnue X_3, ajouter $X_3 = X_1 + X_2$ à E et associer X_3 au résultat t_3.

Exemple

Code	ϕ	E
	$\texttt{a} \mapsto X_a;\ \texttt{i} \mapsto X_i$	$\emptyset$
`t1 = i * 4;`		
	$\texttt{a} \mapsto X_a;\ \texttt{i} \mapsto X_i;\ \texttt{t1} \mapsto X_1$	$\{X_1 = X_i * 4\}$
`t2 = a + t1;`		
	$\texttt{a} \mapsto X_a;\ \texttt{i} \mapsto X_i;\ \texttt{t1} \mapsto X_1$ $\texttt{t2} \mapsto X_2$	$\{X_1 = X_i * 4;$ $X_2 = X_a + X_1\}$
`t6 = i * 4;`		
	$\texttt{a} \mapsto X_a;\ \texttt{i} \mapsto X_i;\ \texttt{t1} \mapsto X_1$ $\texttt{t2} \mapsto X_2;\ \texttt{t6} \mapsto X_1$	$\{X_1 = X_i * 4;$ $X_2 = X_a + X_1\}$
`t7 = a + t6;`		
	$\texttt{a} \mapsto X_a;\ \texttt{i} \mapsto X_i;\ \texttt{t1} \mapsto X_1$ $\texttt{t2} \mapsto X_2;\ \texttt{t6} \mapsto X_1;\ \texttt{t7} \mapsto X_2$	$\{X_1 = X_i * 4;$ $X_2 = X_a + X_1\}$

Traitement du contrôle

Technique standard : analyse de flot sur un graphe de flot de contrôle.



- Aux points de fourche : on propage (ϕ, E) sur les 2 branches.
- Aux points de jointure : on calcule une “intersection” de (ϕ_1, E_1) et (ϕ_2, E_2) (coûteux !), ou bien on repart d’un état symbolique vide.

Optimiser le code à l'aide des résultats de l'analyse statique

Soit une instruction $t_3 = t_1 + t_2$ telle que :

- ϕ associe les inconnues X_1, X_2 à t_1, t_2
- E contient déjà une équation $X = X_1 + X_2$
- il existe une variable t associée à X par ϕ

Alors, on peut remplacer l'instruction par $t_3 = t$
(réutilisation de la valeur calculée antérieurement).

Exemple

```
t1 = i * 4;          --->   t1 = i * 4;          --->   t1 = i * 4;
t2 = a + t1;                t2 = a + t1;                t2 = a + t1;
x = load(t2);               x = load(t2);               x = load(t2);
t3 = j * 4;                 t3 = j * 4;                 t3 = j * 4;
t4 = a + t4;                t4 = a + t4;                t4 = a + t4;
t5 = load(t4);              t5 = load(t4);              t5 = load(t4);
t6 = i * 4;                 t6 = t1;
t7 = a + t6;                t7 = t2;
store(t7, t5);              store(t7, t5);              store(t2, t5);
t8 = j * 4;                 t8 = t3;
t9 = a + t8;                t9 = t4;
store(t9,x);                store(t9,x);                store(t4,x);
```

Preuve de préservation sémantique

Étape 1 : donner une sémantique formelle au langage intermédiaire.

Une espèce de machine abstraite avec comme état :

- p le point de contrôle courant
- e un environnement (variable $\longrightarrow$ valeur)
- m un état mémoire (pointeur $\longrightarrow$ valeur)

et des transitions entre états correspondant à l'exécution d'une instruction élémentaire.

Exemple : si l'instruction au point p est $x = y + z$ avec successeur p', on a la transition

$$(p, e, m) \xrightarrow{\epsilon} (p', e[x \leftarrow e(y) + e(z) \bmod 2^{32}], m)$$

Preuve de préservation sémantique

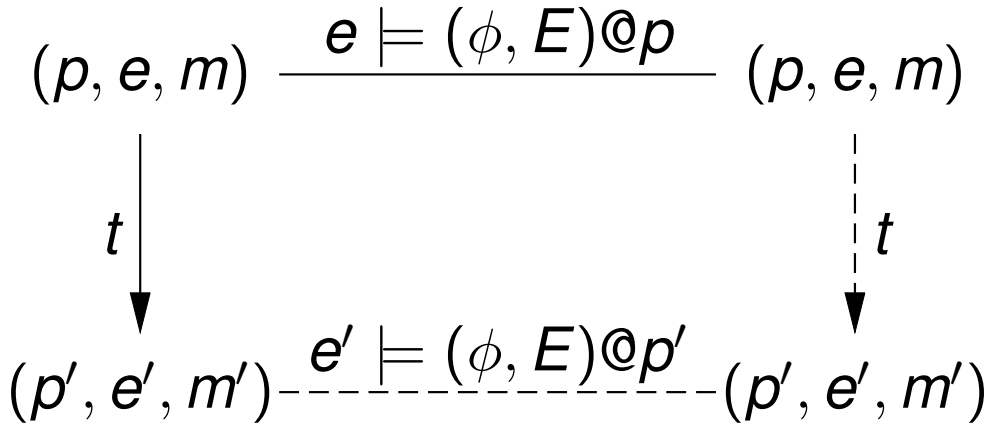
Étape 2 : donner un sens sémantique aux résultats (ϕ, E) de l'analyse statique de *value numbering*.

$e \models (\phi, E)$ s'il existe une valuation ρ : inconnues $\rightarrow$ valeurs telle que

- Si $\phi(x) = X$ alors $e(x) = \rho(X)$
- Si $(X_1 = X_2 + X_3) \in E$ alors $\rho(X_1) = \rho(X_2) + \rho(X_3)$ mod 2^{32}.

Preuve de préservation sémantique

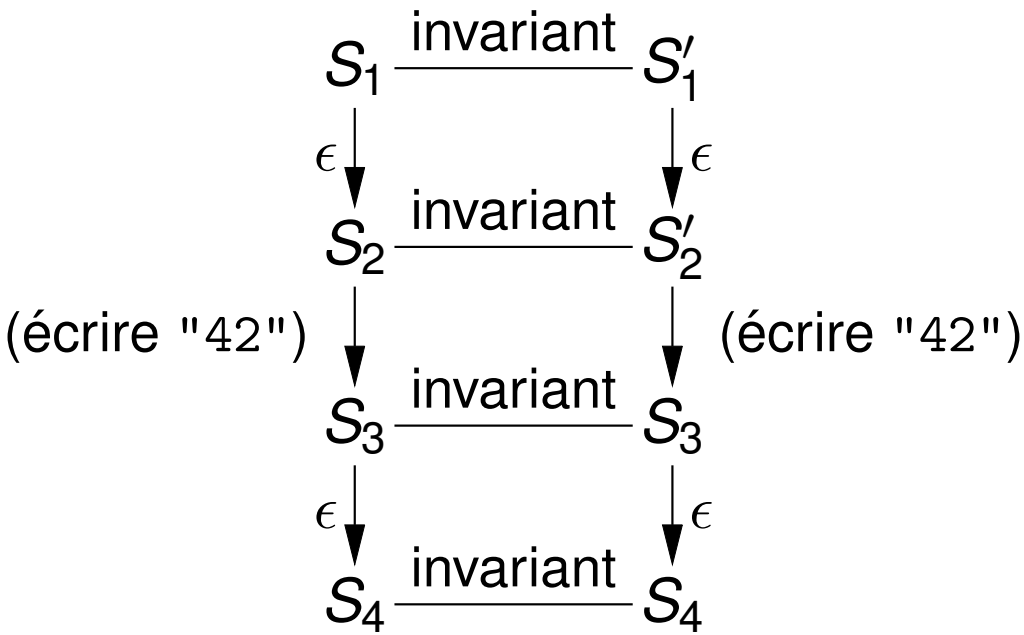
Étape 3 : montrer que toute transition lors de l'exécution du programme d'origine est simulée par une ou plusieurs transitions de l'exécution du programme optimisé.



Programme d'origine Programme optimisé

Preuve de préservation sémantique

Dernière étape : raisonner sur les exécutions complètes des programmes, vues comme des séquences de transitions.



Perspectives

Résultats

L'expérience CompCert n'est pas finie, mais montre déjà que :

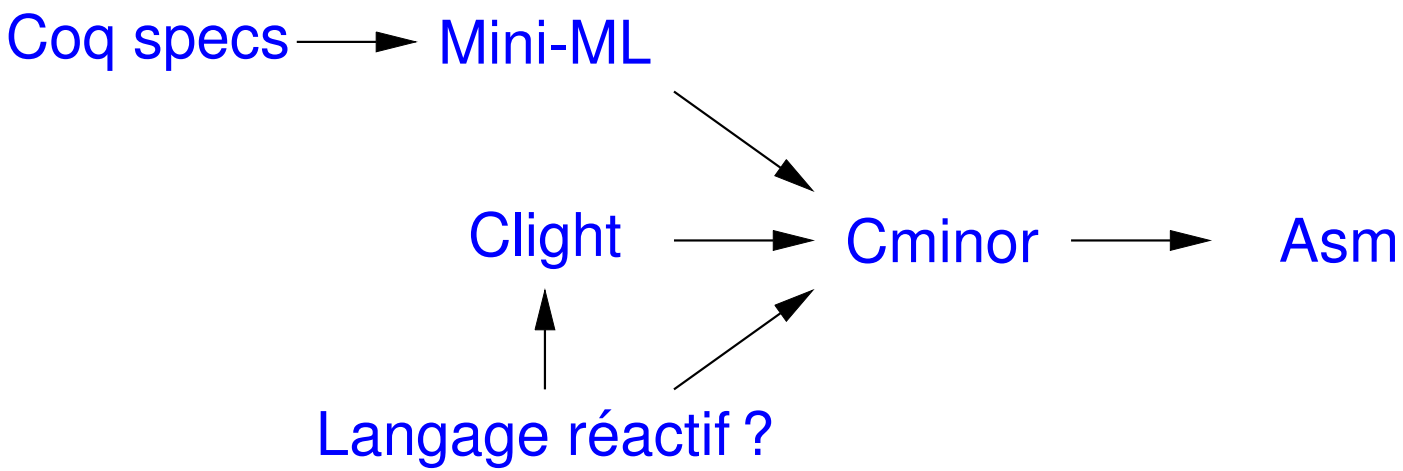
- oui, on peut vérifier formellement un compilateur réaliste ;
- oui, on peut le faire avec les outils de preuve disponibles aujourd'hui.

Qu'avons-nous prouvé, exactement ?

La preuve n'elimine pas toutes les sources d'incertitudes sur le compilateur, mais les réduit à la base de confiance suivante :

- Les sémantiques du langage source (Clight) et du langage cible (assembleur PowerPC et ARM).
- Les parties non prouvées du système :
  - En amont : le parseur/simplifieur CIL.
  - En aval : l'assembleur, le linker, le hardware.
- La chaîne d'exécution du compilateur :
  extraction Coq $\rightarrow$ Caml, compilateur Caml, hardware.
- La logique de Coq et son implémentation.

Extensions 1 : autres langages source



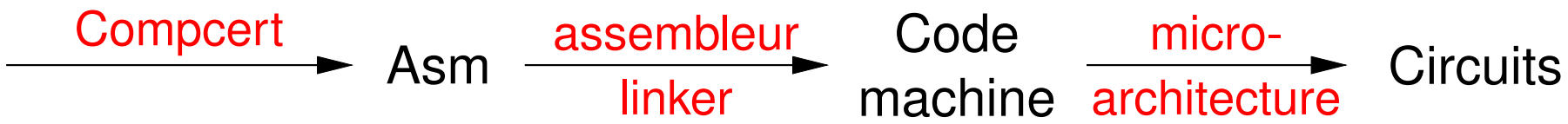
Thèse de Z. Dargaye : compilateur vérifié Mini-ML $\longrightarrow$ Cminor.

Principale difficulté : prouver le *run-time system* (allocateur mémoire et GC), et interfacer cette preuve avec celle du compilateur.

Thèse de S. Glondu : extraction vérifiée Coq $\longrightarrow$ Mini-ML.

Extensions 2 : se rapprocher du hardware

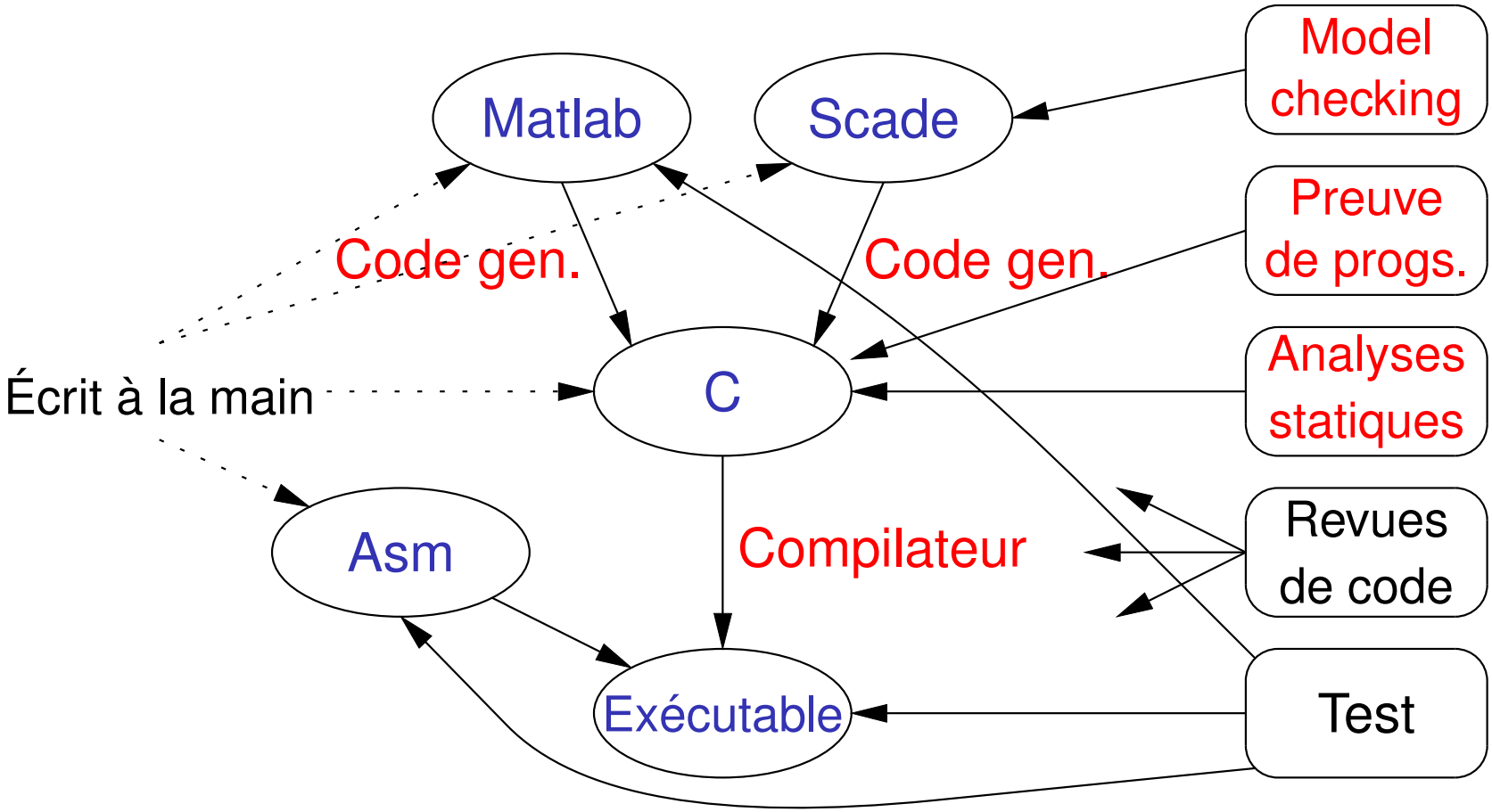
Jeter un pont entre vérification de compilateurs et vérification de processeurs.



Des travaux encourageants dans cette direction :

- De l'assembleur Piton aux *netlist* d'un processeur "maison" (J. Strother Moore et al, 1996)
- Du code machine ARM à la micro-architecture ARM6 (Anthony Fox, U. Cambridge, 2003)

Extensions 3 : vers un environnement de confiance pour le développement de logiciels critiques



Vérifier d'autres outils que le compilateur, ainsi que leur cohérence. . .