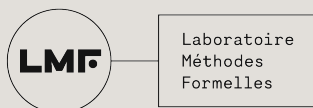


Des briques de calcul formel plus solides avec Capla

JFLA 2025

30 Janvier 2025

Josué Moreau



Exemples de bibliothèques largement utilisées :

- ▶ GMP : opérations sur des grands entiers
- ▶ BLAS : opérations sur des vecteurs et matrices
- ▶ FFTW : transformée rapide de Fourier

Exemples de bibliothèques largement utilisées :

- ▶ GMP : opérations sur des grands entiers
- ▶ BLAS : opérations sur des vecteurs et matrices
- ▶ FFTW : transformée rapide de Fourier

Erreur d'implémentation (GMP $\leq$ 5.1.1)

`mpz_pown_ui(r, b, e, m) :  $r \leftarrow b^e \bmod m$`

Résultat incorrect quand b fait plus de 15000 chiffres.

Erreur de compilation (GMP 6.2.0)

Crash et erreurs de calculs avec Xcode 11 à 11.3 sous MacOS.

- Écrites en C et Fortran

- ▶ Écrites en C et Fortran
- ▶ Avantages :
 - Modèle mémoire « simple » et performances prédictibles
 - Compilateurs très optimisants

- ▶ Écrites en C et Fortran
- ▶ Avantages :
 - Modèle mémoire « simple » et performances prédictibles
 - Compilateurs très optimisants
- ▶ Inconvénients :
 - Langages non sûrs
 - Preuve de programme difficile (les preuves sont déjà mathématiquement difficiles)
 - Compilateurs trop compliqués $\Rightarrow$ confiance limitée dans le code généré

- ▶ Un langage dédié à l'écriture des algorithmes des bibliothèques de calcul formel :
 - opérations de bas niveau sur les tableaux, compilation efficace
 - sûr (pas de comportements indéfinis)
 - avec une sémantique simplifiant la preuve de programmes
- ▶ Un compilateur, formellement vérifié avec Rocq
 - utilisant CompCert comme backend
 - avec préservation de la sémantique
 - sûreté du typage et type-checker vérifié

BLAS : produit scalaire (Fortran)

```
complex*16 function zdotu(n, zx, zy, incx, incy)
  integer incx, incy, n
  complex*16 zx(*), zy(*)
```

► Documentation : la taille de zx est $1 + (n - 1) \cdot |incx|$.

GMP : élévation au carré (C)

```
void mpn_sqr(mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n)
```

► Documentation :

- Les tailles de rp et $s1p$ sont respectivement $2n$ et n .
- Pas de superposition entre rp et $s1p$.

Exemple: multiplication de matrices

```
fun mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p], m n p: u64) {  
  for i = 0 .. m  
    for j = 0 .. p {  
      dest[i, j] = 0;  
      for k = 0 .. n {  
        dest[i, j] = dest[i, j] + a[i, k] * b[k, j];  
      }  
    }  
}
```

Exemple: multiplication de matrices

Tailles explicites en paramètres

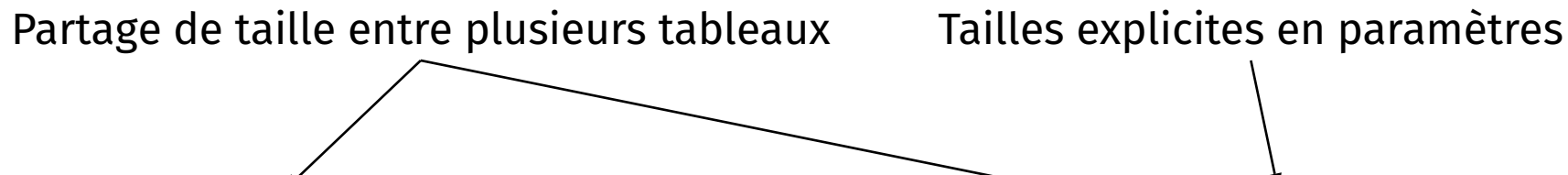


```
fun mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p], m n p: u64) {  
  for i = 0 .. m  
    for j = 0 .. p {  
      dest[i, j] = 0;  
      for k = 0 .. n {  
        dest[i, j] = dest[i, j] + a[i, k] * b[k, j];  
      }  
    }  
}
```

Exemple: multiplication de matrices

Partage de taille entre plusieurs tableaux

Tailles explicites en paramètres



```
fun mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p], m n p: u64) {  
  for i = 0 .. m  
    for j = 0 .. p {  
      dest[i, j] = 0;  
      for k = 0 .. n {  
        dest[i, j] = dest[i, j] + a[i, k] * b[k, j];  
      }  
    }  
}
```

Exemple: multiplication de matrices

Partage de taille entre plusieurs tableaux

Tailles explicites en paramètres

```
fun mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p], m n p: u64) {  
  for i = 0 .. m  
    for j = 0 .. p {  
      dest[i, j] = 0;  
      for k = 0 .. n {  
        dest[i, j] = dest[i, j] + a[i, k] * b[k, j];  
      }  
    }  
}
```

Distinction tableaux mutables / persistants

Exemple: multiplication de matrices

Partage de taille entre plusieurs tableaux

Tailles explicites en paramètres

```
fun mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p], m n p: u64) {
  for i = 0 .. m
    for j = 0 .. p {
      dest[i, j] = 0;
      for k = 0 .. n {
        dest[i, j] = dest[i, j] + a[i, k] * b[k, j];
      }
    }
}
```

Pas de comportements
indéfinis pour les accès
aux tableaux (tests dynamiques)

Distinction tableaux mutables / persistants

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
  t[1] = 5;  
  let s: [i64; 3] = alloc;  
}
```

```
fun f(...) {  
→   ...  
   g(u, t, n);  
   ...  
}
```

f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 2}$
---	---

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
→ fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
    t[1] = 5;  
    let s: [i64; 3] = alloc;  
}  
  
fun f(...) {  
    ...  
    g(u, t, n);  
    ...  
}
```

g	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 2}$
f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 2}$

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
  → t[1] = 5;  
  let s: [i64; 3] = alloc;  
}  
  
fun f(...) {  
  ...  
  g(u, t, n);  
  ...  
}
```

g	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 5}$
f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 2}$

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
  t[1] = 5;  
  let s: [i64; 3] = alloc;  
→ }  
  
fun f(...) {  
  ...  
  g(u, t, n);  
  ...  
}
```

g	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 5}, s \mapsto \boxed{0 \mid 0 \mid 0}$
f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 2}$

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
  t[1] = 5;  
  let s: [i64; 3] = alloc;  
}
```

```
fun f(...) {  
  ...  
→  g(u, t, n);  
  ...  
}
```

f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 5}$
---	---

Les programmes se comportent comme si les tableaux étaient copiés à l'entrée et à la sortie de fonction.

```
fun g(u: [i64; n], t: mut [i64; n], n: u64) {  
  t[1] = 5;  
  let s: [i64; 3] = alloc;  
}
```

```
fun f(...) {  
  ...  
→  g(u, t, n);  
  ...  
}
```

f	$n \mapsto 2, u \mapsto \boxed{4 \mid 6}, t \mapsto \boxed{1 \mid 5}$
---	---

- Raisonnements sur les programmes plus simples
- mais politique de non-alias nécessaire pour garantir un code généré efficace

Structured values: $v ::= \text{Vint } n \mid \dots \mid \text{Varr } [v_1, v_2, \dots]$

Local environments: $E : x \mapsto v$

$$\begin{array}{c}
 \text{perm}(t) \geq \text{Mutable} \quad E \vdash e \Rightarrow \text{Vint}_{64} n \quad n < \text{sz}_t \quad \dots \\
 E \vdash e' \Rightarrow v \quad \text{primitive}(v) \\
 \hline
 (E, \boxed{t[e] = e'}, k) \rightarrow (E[t \leftarrow E(t)[n \leftarrow v]], \boxed{}, k)
 \end{array}
 \text{WRITE}$$

$\text{Owned} \geq \text{Mutable} \geq \text{Shared}$

typage

exécution

Appel de fonction:

$$E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in \mathbf{f.sig_args}$$

$$\mathbf{f.params} = \vec{x} \quad E_{\mathbf{f}} = \mathbf{build_env}(\vec{x}, \vec{v})$$

...

$$\frac{}{(E, \boxed{y = \mathbf{f}(\vec{a})}, k) \rightarrow (E_{\mathbf{f}}, \mathbf{f.body}, \mathbf{Kreturnto}(y, E, m, k))} \text{CALL}$$

Appel de fonction:

$$\begin{aligned}
 &E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in f.\text{sig_args} \\
 &f.\text{params} = \vec{x} \quad E_f = \text{build_env}(\vec{x}, \vec{v}) \\
 &\forall i, f.\text{perm}(x_i) \leq \text{perm}(p_i)
 \end{aligned}$$

...

$$\frac{}{(E, \boxed{y = f(\vec{a})}, k) \rightarrow (E_f, f.\text{body}, \text{Kreturnto}(y, E, m, k))} \text{CALL}$$

Appel de fonction:

$$\begin{array}{c} E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in f.\text{sig_args} \\ f.\text{params} = \vec{x} \quad E_f = \text{build_env}(\vec{x}, \vec{v}) \\ \forall i, f.\text{perm}(x_i) \leq \text{perm}(p_i) \\ \forall i \, j, f.\text{perm}(x_i) \geq \text{Mutable} \wedge i \neq j \Rightarrow p_i \not\leq p_j \wedge p_j \not\leq p_i \\ \dots \\ \hline (E, \boxed{y = f(\vec{a})}, k) \rightarrow (E_f, f.\text{body}, \text{Kreturnto}(y, E, m, k)) \quad \text{CALL} \end{array}$$

Appel de fonction:

$$\begin{array}{c}
 E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in f.\text{sig_args} \\
 f.\text{params} = \vec{x} \quad E_f = \text{build_env}(\vec{x}, \vec{v}) \\
 \forall i, f.\text{perm}(x_i) \leq \text{perm}(p_i) \\
 \forall i \, j, f.\text{perm}(x_i) \geq \text{Mutable} \wedge i \neq j \Rightarrow p_i \not\leq p_j \wedge p_j \not\leq p_i \\
 \text{valid_call}(E, f, \vec{p}) \\
 \dots \\
 \hline
 (E, \boxed{y = f(\vec{a})}, k) \rightarrow (E_f, f.\text{body}, \text{Kreturnto}(y, E, m, k)) \quad \text{CALL}
 \end{array}$$

Appel de fonction:

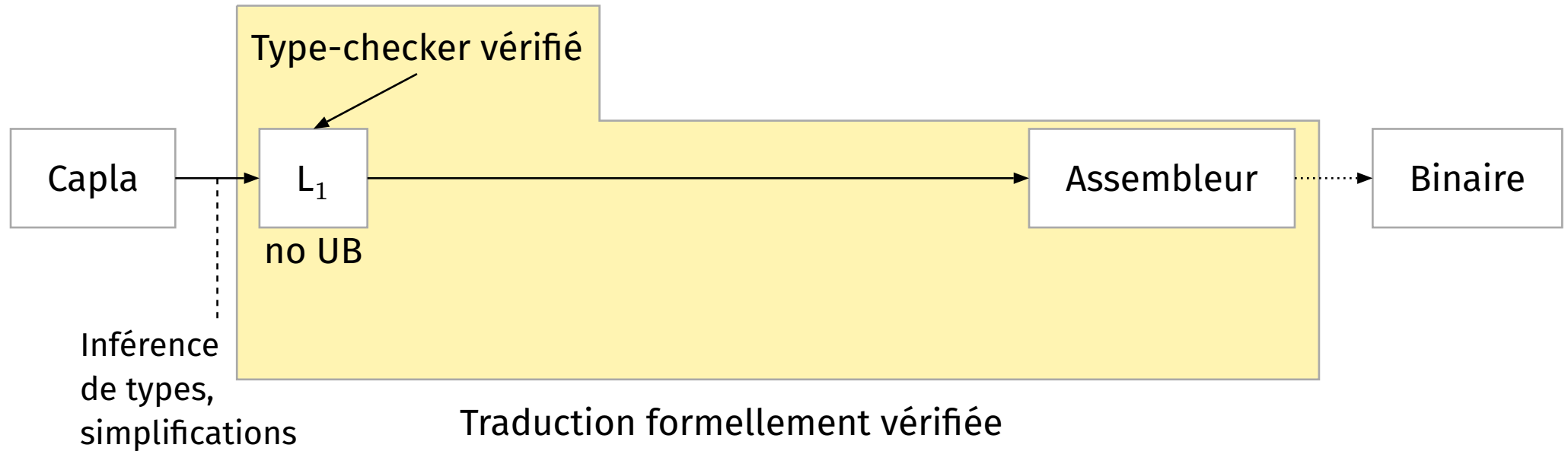
$$\begin{array}{c}
 E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in f.\text{sig_args} \\
 f.\text{params} = \vec{x} \quad E_f = \text{build_env}(\vec{x}, \vec{v}) \\
 \forall i, f.\text{perm}(x_i) \leq \text{perm}(p_i) \\
 \forall i \ j, f.\text{perm}(x_i) \geq \text{Mutable} \wedge i \neq j \Rightarrow p_i \not\leq p_j \wedge p_j \not\leq p_i \\
 \text{valid_call}(E, f, \vec{p}) \\
 m = \{(p_i, x_i) \mid f.\text{perm}(x_i) = \text{Mutable}\} \\
 \dots \\
 \hline
 (E, \boxed{y = f(\vec{a})}, k) \rightarrow (E_f, f.\text{body}, \text{Kreturnto}(y, E, m, k)) \quad \text{CALL}
 \end{array}$$

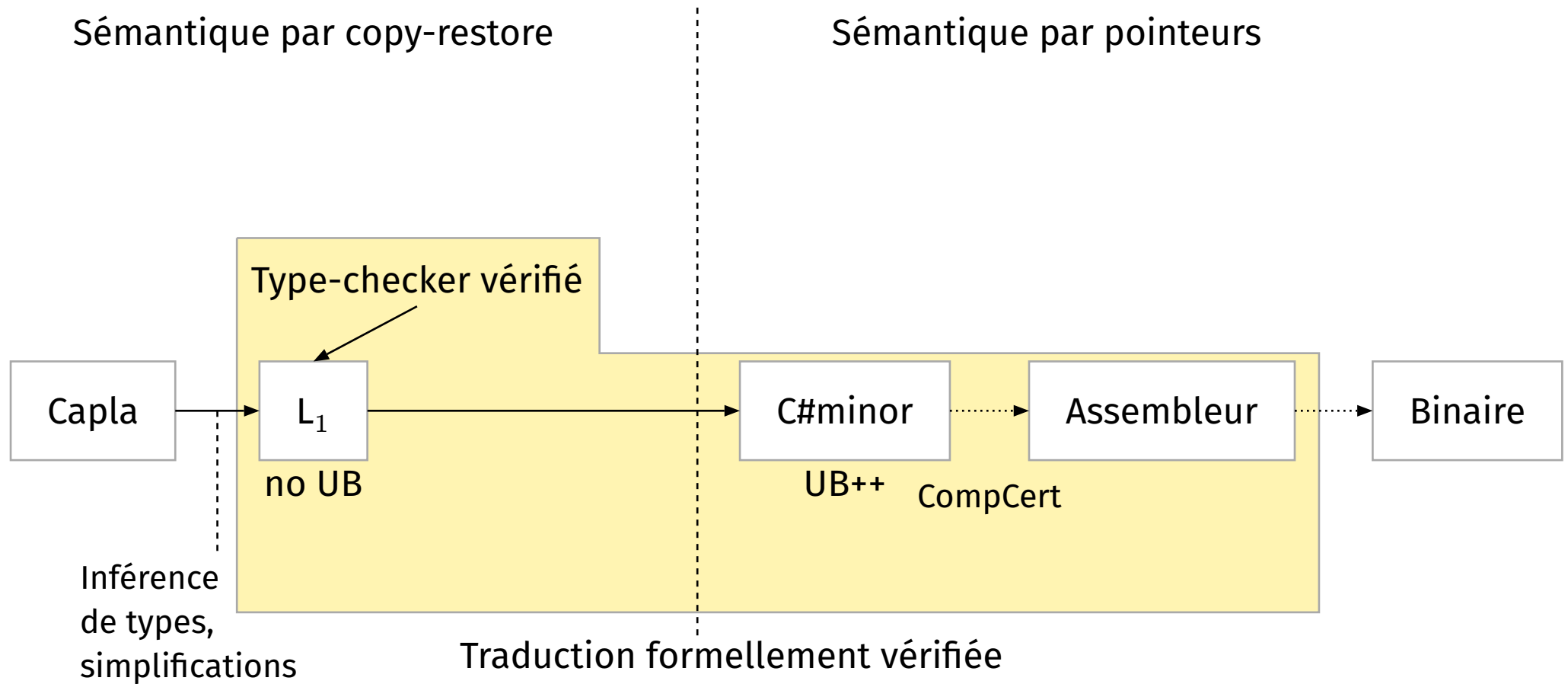
Appel de fonction:

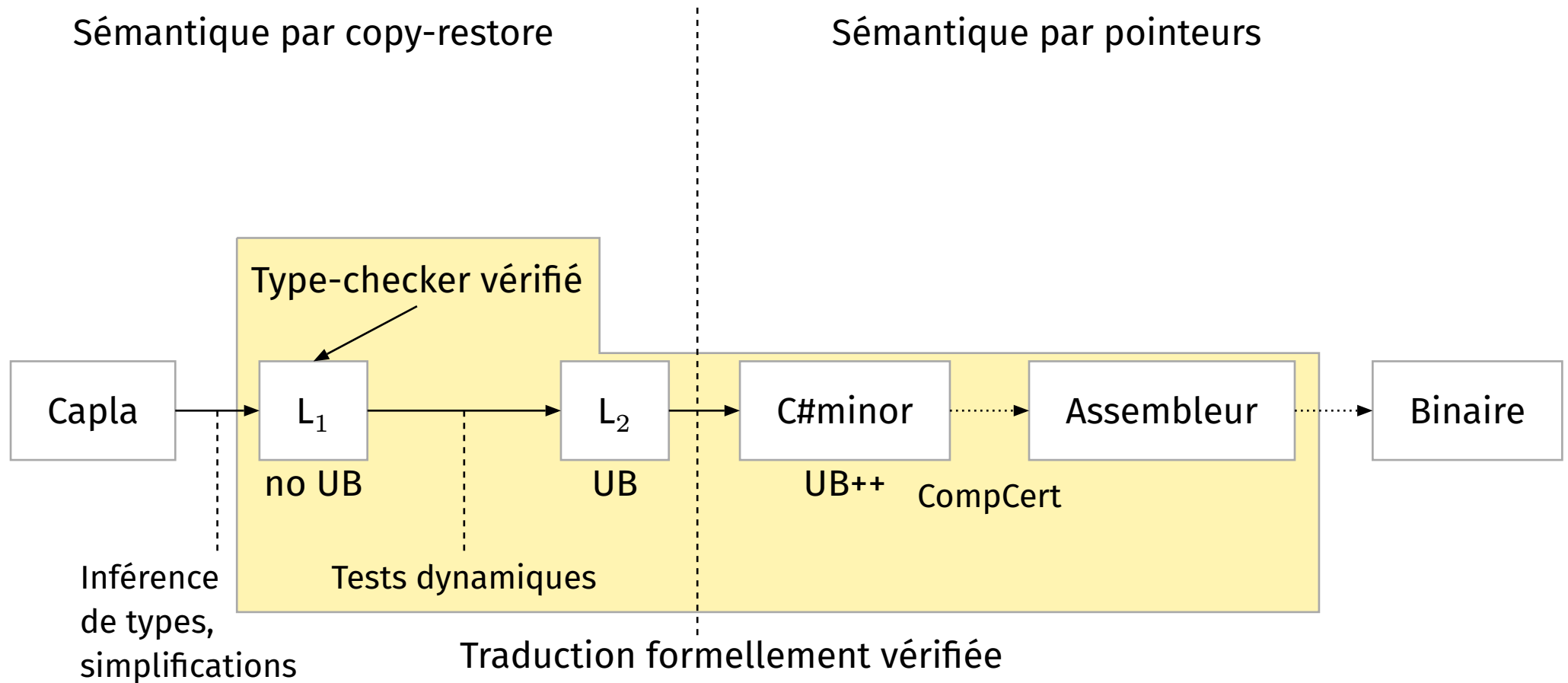
$$\begin{array}{c}
 E \vdash \vec{a} \Rightarrow \vec{p} \quad E(\vec{p}) = \vec{v} \quad \vec{v} \in \text{f.sig_args} \\
 \text{f.params} = \vec{x} \quad E_{\text{f}} = \text{build_env}(\vec{x}, \vec{v}) \\
 \forall i, \text{f.perm}(x_i) \leq \text{perm}(p_i) \\
 \forall i \ j, \text{f.perm}(x_i) \geq \text{Mutable} \wedge i \neq j \Rightarrow p_i \not\leq p_j \wedge p_j \not\leq p_i \\
 \text{valid_call}(E, \text{f}, \vec{p}) \\
 m = \{(p_i, x_i) \mid \text{f.perm}(x_i) = \text{Mutable}\} \\
 \dots \\
 \hline
 (E, \boxed{y = \text{f}(\vec{a})}, k) \rightarrow (E_{\text{f}}, \text{f.body}, \text{Kreturnto}(y, E, m, k)) \quad \text{CALL}
 \end{array}$$

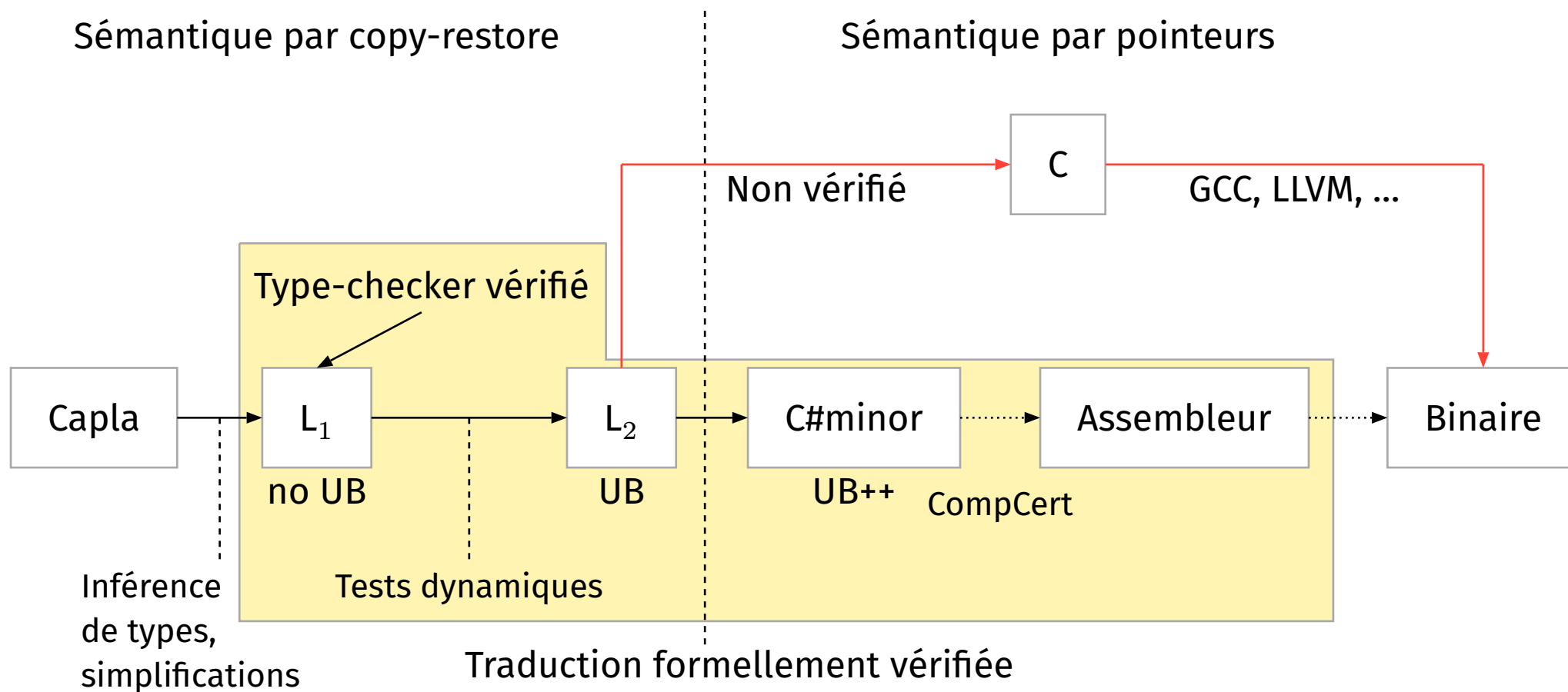
Retour de fonction:

$$\hline
 (E_{\text{f}}, \boxed{\text{return } v}, \text{Kreturnto}(y, E, (\vec{p}, \vec{x}), k)) \rightarrow (E[\vec{p} \leftarrow \overline{E_{\text{f}}[x]}][y \leftarrow v], \boxed{}, k) \quad \text{RETURN}$$









Deuxième exemple: produit scalaire (BLAS)

```
fun zdotu(n: i32, zx: [f64; 1 + (n - 1) * incx, 2], incx: i32,
          zy: [f64; 1 + (n - 1) * incy, 2], incy: i32,
          res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;

  if incx == 1 && incy == 1 {
    for i: i32 = 0 .. n {
      res[0] = res[0] + (zx[i, 0] * zy[i, 0] - zx[i, 1] * zy[i, 1]);
      res[1] = res[1] + (zx[i, 1] * zy[i, 0] + zx[i, 0] * zy[i, 1]);
    }
  } else {
    let ix: i32 = 0, iy: i32 = 0;
    if (incx < 0) ix = (-n+1)*incx;
    if (incy < 0) iy = (-n+1)*incy;
    ...
  }
}
```

Deuxième exemple: produit scalaire (BLAS)

```

fun zdotu(n: i32, zx: [f64; 1 + (n - 1) * incx, 2], incx: i32,
          zy: [f64; 1 + (n - 1) * incy, 2], incy: i32,
          res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;

  if incx == 1 && incy == 1 {
    for i: i32 = 0 .. n {
      res[0] = res[0] + (zx[i, 0] * zy[i, 0] - zx[i, 1] * zy[i, 1]);
      res[1] = res[1] + (zx[i, 1] * zy[i, 0] + zx[i, 0] * zy[i, 1]);
    }
  } else {
    let ix: i32 = 0, iy: i32 = 0;
    if (incx < 0) ix = (-n+1)*incx;
    if (incy < 0) iy = (-n+1)*incy;
    ...
  }
}

```

Test dynamique: $1 < 2$



Test dynamique: $i < 1 + (n - 1) \cdot incy$



Deuxième exemple: produit scalaire (BLAS)

```

fun zdotu(n: i32, zx: [f64; 1 + (n - 1) * incx, 2], incx: i32,
          zy: [f64; 1 + (n - 1) * incy, 2], incy: i32,
          res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;

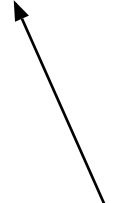
  if incx == 1 && incy == 1 {
    for i: i32 = 0 .. n {
      res[0] = res[0] + (zx[i, 0] * zy[i, 0] - zx[i, 1] * zy[i, 1]);
      res[1] = res[1] + (zx[i, 1] * zy[i, 0] + zx[i, 0] * zy[i, 1]);
    }
  } else {
    let ix: i32 = 0, iy: i32 = 0;
    if (incx < 0) ix = (-n+1)*incx;
    if (incy < 0) iy = (-n+1)*incy;
    ...
  }
}

```

Test dynamique: $1 < 2$
Éliminé trivialement



Test dynamique: $i < 1 + (n - 1) \cdot incy$
Éliminé, mais le compilateur a besoin d'aide
`assert(1 + (n - 1) * incy == n)`



	CompCert	GCC	LLVM	GCC avec assertions	LLVM avec assertions
zdotu	2.25	1.01	0.93	1.06	0.82
saxpy	5.37	4.25	0.78	3.68	0.78
sgemv	2.86	1.32	0.96	1.32	0.87
dgemv	1.73	1.03	0.59	0.68	0.53
dtrsv (N)	2.87	1.32	1.78	0.92	1.60
dtrsv (T)	2.30	1.14	1.35	0.91	0.90
mpn_addmul_1	2.74	1.20	1.06	—	—

- ▶ ≥ 1 : plus lent que l'implémentation de référence
- ▶ Les entrées sont suffisamment petites pour minimiser les défauts de caches
- ▶ x86-64 BLAS/LAPACK 3.12.0 (Fortran) et GMP 6.3.0 (assembleur écrit à la main)
- ▶ LLVM 17.0.6 et GCC 13.2.0, avec options `-O2 -ftree-vectorize`

Invariants

- 1 Le contenu de l'environnement a le bon type.
- 2 Les variables nécessaires sont initialisées.
- 3 Les tailles de tableaux sont valides (*i.e.*, au plus la taille effective du tableau).

Théorème (Sûreté du typage)

*Soit un programme L_1 bien typé et un état non-final s ,
si tous les invariants sont respectés par s , alors*

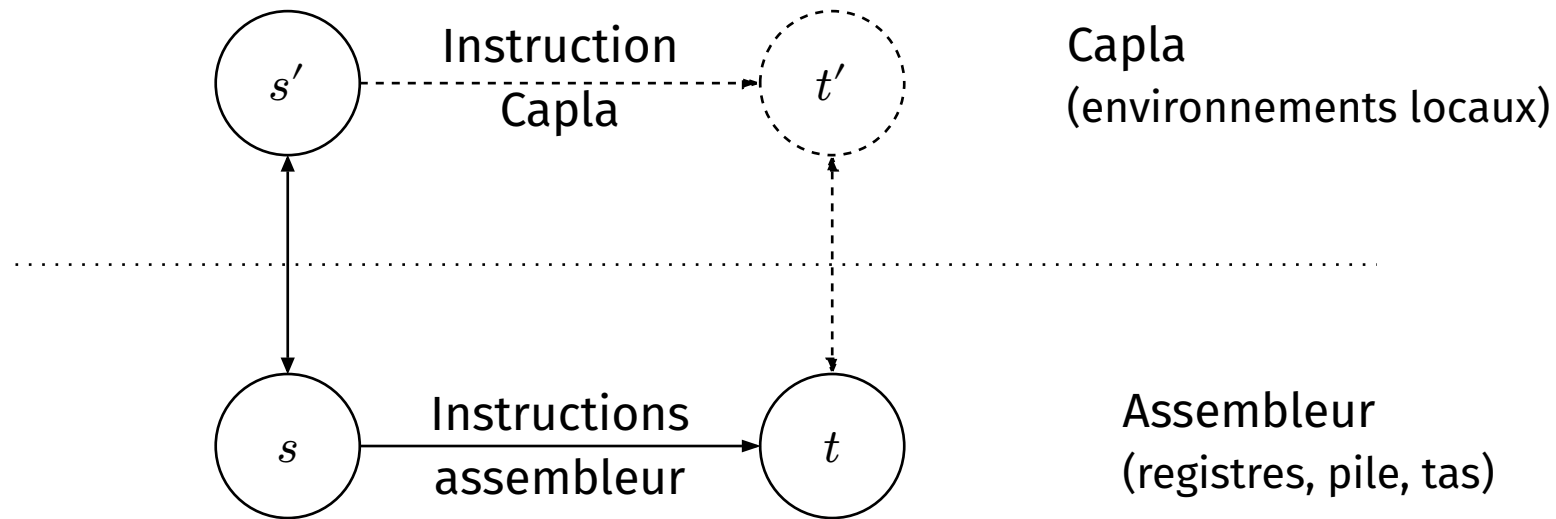
- *il existe un état t tel que $s \rightarrow t$ et,*
- *pour tout état t tel que $s \rightarrow t$, t respecte les invariants.*



Théorème (Correction du compilateur - simulation arrière)

Soit $s \xrightarrow{ASM} t$, pour tout état s' de L_1 tel que $s' \leftrightarrow s$:

- $|t| < |s|$
- ou il existe t' tel que $s' \xrightarrow{L_1} t'$ et $t' \leftrightarrow t$.



► Langage sûr pour le calcul formel de bas niveau

■ Politique de non-alias pour la mémoire mutable

■ Sémantique simple pour la preuve de programmes

► Compilateur formellement vérifié vers l'assembleur

■ Preuves de correction et de sûreté vérifiées avec Rocq

~ 5000 lignes de code, ~ 8000 lignes de spec, ~ 10000 lignes de preuve

► Sortie C non vérifiée

```
// BLAS triangular system solving
fun dtrsv(uplo trans diag: u8, n: i32,
  a:      [f64; n, lda], lda: i32,
  x: mut [f64; 1+(n-1)*incx], incx: i32)
```

Travaux en cours :

► Logique de programmes et calcul de WP

► Vues (arithmétique de pointeurs): `let [u: .. k; v: ..] = t`