A safe low-level language for computer algebra and its formally verified compiler

Cambium seminar

June 13, 2025

Guillaume Melquiond

LMF Laboratoire Méthodes Formelles

université PARIS-SACLAY

Josué Moreau

Inria

erc

Reliability of computer algebra libraries

Examples of widely used libraries:

- GMP: operations on arbitrarily large integers
- BLAS: operations on vectors and matrices
- FFTW: fast Fourier transform

Reliability of computer algebra libraries

Examples of widely used libraries:

- GMP: operations on arbitrarily large integers
- BLAS: operations on vectors and matrices
- FFTW: fast Fourier transform

Bug (GMP $\leq$ 5.1.1)

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

Computes garbage if `b` is over 15000 decimals.

Bug (GMP 6.2.0)

MacOS Xcode 11 prior to 11.3 miscompiles GMP, leading to crashes and miscomputation.

Implementations of computer algebra libraries

- Libraries written in C and Fortran (and handwritten assembly)

Implementations of computer algebra libraries

- Libraries written in C and Fortran (and handwritten assembly)
- Pros:
  - Do not get in the way of the user
  - Highly optimizing compilers

Implementations of computer algebra libraries

- Libraries written in C and Fortran (and handwritten assembly)
- Pros:
  - Do not get in the way of the user
  - Highly optimizing compilers
- Cons:
  - These languages are not safe
  - Difficult program verification (programs are already mathematically difficult to prove)
  - Compilers are too complicated for a total confidence in the generated code

Capla

- A language dedicated to computer algebra libraries:
  - low-level array manipulations, compilable to efficient code
  - safe (no undefined behaviors)
  - with semantics making the proof of programs simpler
- A compiler, formally verified with Rocq
  - using CompCert as backend
  - semantics preservation
  - type safety and verified type-checker

Table of contents

Language

Expressiveness in the signature of functions

BLAS dot product (Fortran)

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

- Documentation states that size of `zx` is $1 + (\texttt{n} - 1) \cdot |\texttt{incx}|$.

GMP square (C)

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

- Documentation states:
  - Size of `rp` and `s1p` are respectively $2\texttt{n}$ and `n`.
  - No overlap between `rp` and `s1p`.

Example: multiplication of 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];
      }
    }
}
```

Example: multiplication of matrices

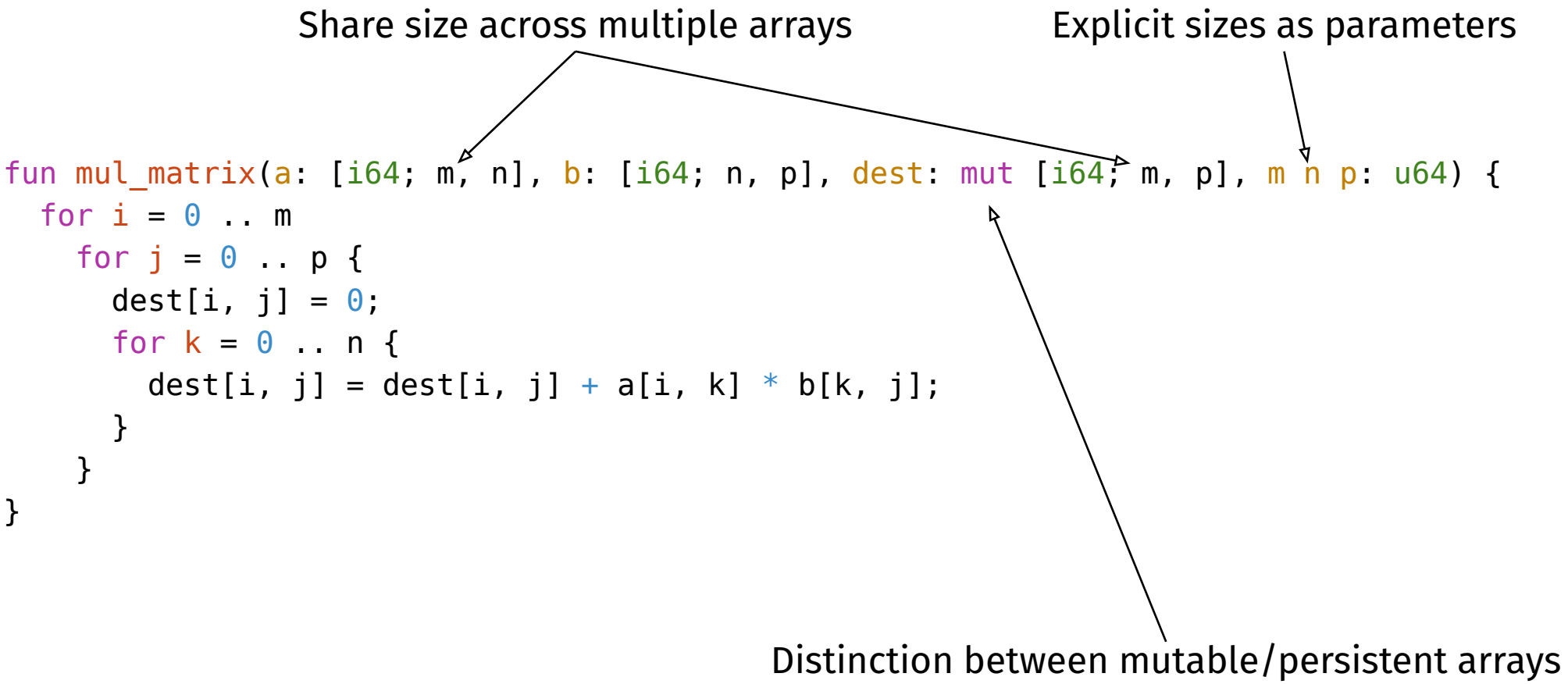
Explicit sizes as parameters

```
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];
      }
    }
}
```

Example: multiplication of 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];
      }
    }
}
```

Example: multiplication of 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];
      }
    }
}
```

Example: multiplication of matrices

Share size across multiple arrays

Explicit sizes as parameters

```
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];
      }
    }
}
```

No undefined behavior for array accesses (dynamic check)

Distinction between mutable/persistent arrays

Runtime checks

```
fun f(t: mut [i64; n], n k: u64)
    -> u64 {

  t[2] = 7;

  return n / k;
}
```

Runtime checks

```
fun f(t: mut [i64; n], n k: u64)
    -> u64 {

  t[2] = 7;

  return n / k;
}
```

→

```
uint64_t f(int64_t* restrict t,
           uint64_t n, uint64_t k) {
  assert(2 < n);
  t[2] = 7;
  assert(k != 0);
  return (n / k);
}
```

Second example: Complex dot product in 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;
    ...
  } }
```

Second example: Complex dot product in 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;
    ...
  } }
```

Dynamic test: $1 < 2$

Dynamic test: $\texttt{i} < 1 + (\texttt{n} - 1) \cdot \texttt{incy}$

Second example: Complex dot product in 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;
    ...
  } }
```

Dynamic test: $1 < 2$
Trivially eliminated

Dynamic test: $\mathtt{i} < 1 + (\mathtt{n} - 1) \cdot \mathtt{incy}$
Eliminated but the compiler needs a bit of help

Aliasing

- No aliasing between a mutable array and any other array
  - Expressive signatures: non-mutable variables are trivially unmodified

```
fun f(a: [i64; 1], b: mut [i64; 1]) {
  let v = a[0];
  b[0] = 4;
  assert (a[0] == v); // Ok
}
```

Aliasing

- No aliasing between a mutable array and any other array
  - Expressive signatures: non-mutable variables are trivially unmodified
  - Copy-restore semantics: no global memory, only local environments

```
fun f(a: [i64; 1], b: mut [i64; 1]) {
  let v = a[0];
  b[0] = 4;
  assert (a[0] == v); // Ok
}
```

Aliasing

- No aliasing between a mutable array and any other array
  - Expressive signatures: non-mutable variables are trivially unmodified
  - Copy-restore semantics: no global memory, only local environments
  - Efficient generated programs: use pointers instead of deep copies

```
fun f(a: [i64; 1], b: mut [i64; 1]) {
  let v = a[0];
  b[0] = 4;
  assert (a[0] == v); // Ok
}
```

→

```
void f(int64_t* a, int64_t* restrict b) {
  uint64_t v = a[0];
  b[0] = 4;
}
```

Aliasing

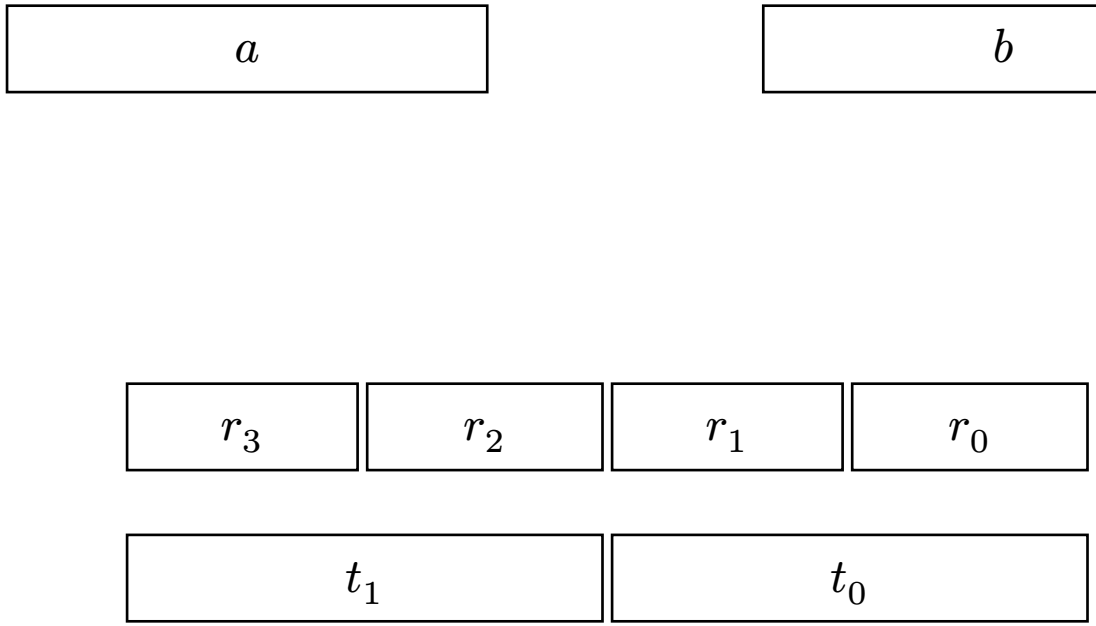
- No aliasing between a mutable array and any other array
  - Expressive signatures: non-mutable variables are trivially unmodified
  - Copy-restore semantics: no global memory, only local environments
  - Efficient generated programs: use pointers instead of deep copies
  - Program verification: no need for separation logic for proofs

```
fun f(a: [i64; 1], b: mut [i64; 1]) {
  let v = a[0];
  b[0] = 4;
  assert (a[0] == v); // Ok
}
```

→

```
void f(int64_t* a, int64_t* restrict b) {
  uint64_t v = a[0];
  b[0] = 4;
}
```

Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

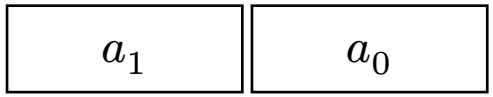


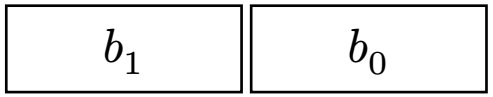
$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$

$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$

Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

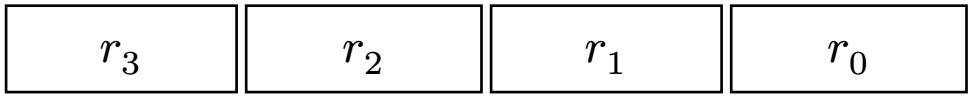


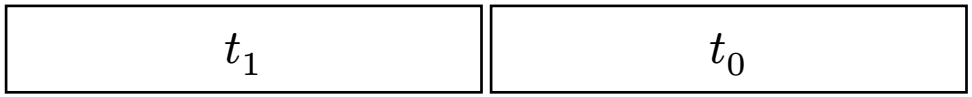


$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

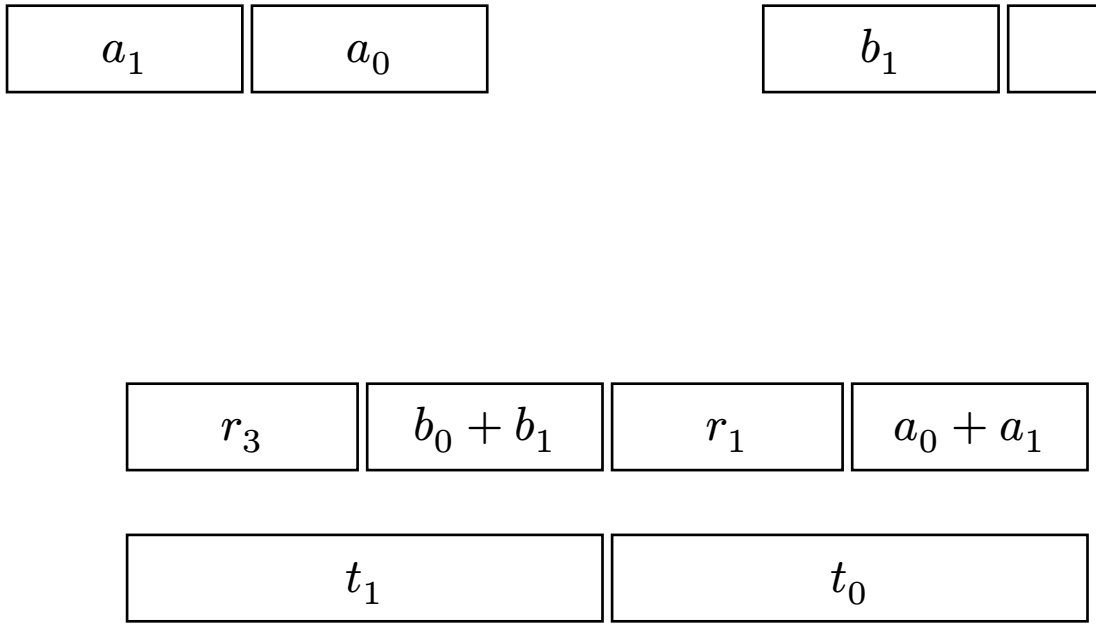
$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$

$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$





Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

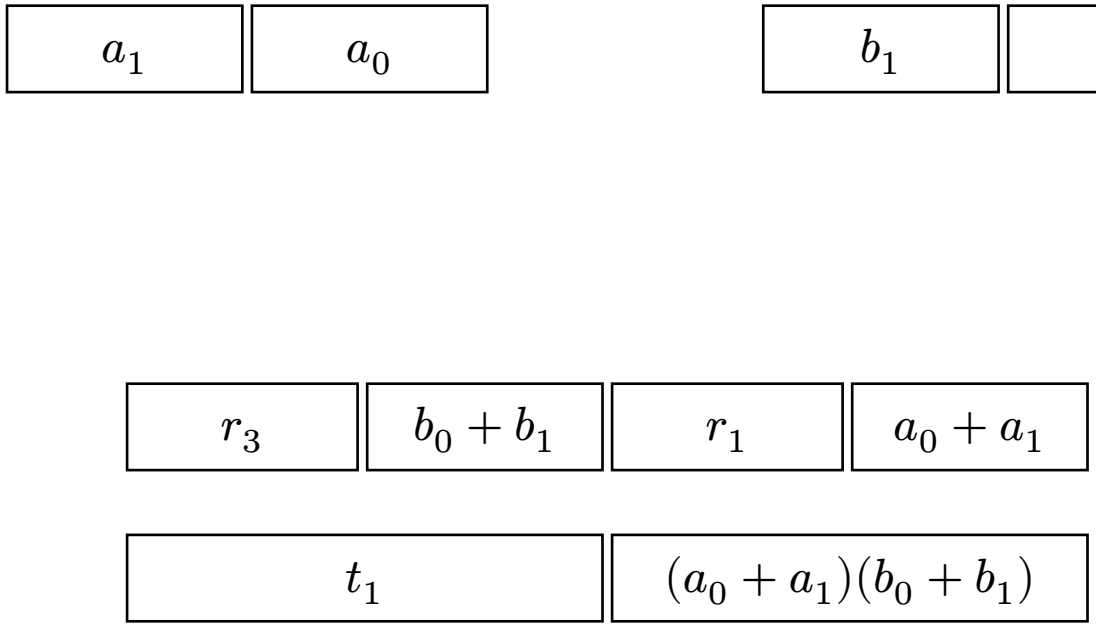


$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$

$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$

Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

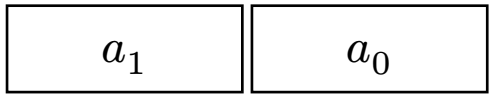


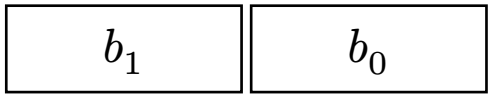
$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$

$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$

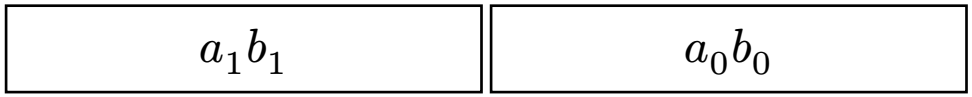
Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

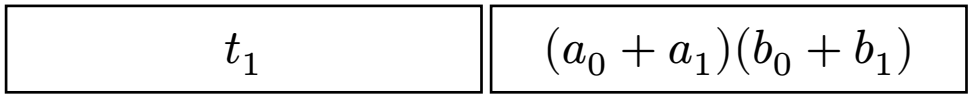




$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$





$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$

Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

```
typedef long long int i64;
typedef unsigned long long int u64;

void karatsuba(i64* r, i64* a, i64* b,
               i64* t, u64 n) {
  ...
  u64 k = n / 2;
  add(r, a, a + k, k);
  add(r + 2 * k, b, b + k, k);
  karatsuba(t, r, r + 2 * k,
            t + 2 * k, k);
  karatsuba(r, a, b, t + 2 * k, k);
  karatsuba(r + 2 * k, a + k, b + k,
            t + 2 * k, k);
  sub(t, t, r, 2 * k);
  sub(t, t, r + 2 * k, 2 * k);
  add(r + k, r + k, t, 2 * k);
}
```

$$r_0 \leftarrow a_0 + a_1$$
$$r_2 \leftarrow b_0 + b_1$$
$$t_0 \leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1)$$

$$r_0 \leftarrow a_0 b_0$$
$$r_2 \leftarrow a_1 b_1$$

$$t_0 \leftarrow t_0 - r_0$$
$$t_0 \leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1$$
$$r_{1,2} \leftarrow r_{1,2} + t_0$$

Non-aliasing policy and pointer arithmetic (Karatsuba algorithm on polynomials)

```
fun karatsuba(r: mut [i64; 2 * n], a b: [i64; n], t: mut [i64; 2 * n], u: u64) {
  ...
  let k = n / 2;
  let [a0: ..k; a1: k..] = a;
  let [b0: ..k; b1: k..] = b;
  let [t0: ..(2 * k); t1: ..] = t;
  { let [r0: ..(2 * k); r2: ..] = r;
    add(r0[..k], a0, a1, k);
    add(r2[..k], b0, b1, k);
    karatsuba(t0, r0[..k], r2[..k], t1, k);
    karatsuba(r0, a0, b0, t1, k);
    karatsuba(r2, a1, b1, t1, k);
    decr(t0, r0, 2 * k);
    decr(t0, r2, 2 * k); }
  incr(r[k..(3 * k)], t0, 2 * k);
}
```

$$
\begin{aligned}
r_0 &\leftarrow a_0 + a_1 \\
r_2 &\leftarrow b_0 + b_1 \\
t_0 &\leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1) \\
r_0 &\leftarrow a_0 b_0 \\
r_2 &\leftarrow a_1 b_1 \\
t_0 &\leftarrow t_0 - r_0 \\
t_0 &\leftarrow t_0 - r_2 = a_1 b_0 + a_0 b_1 \\
r_{1,2} &\leftarrow r_{1,2} + t_0
\end{aligned}
$$

Semantics

Reading into an array

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

Local environments: $E : x \rightharpoonup v$

$$\frac{t \in E \qquad E \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad n < \texttt{sz}_t}{E \vdash t[e] \Rightarrow E(t)[n]}\ \textsc{Read}$$

Reading into an array

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

Local environments: $E : x \rightharpoonup v$

$$\frac{\overbrace{t \in E}^{\text{typing}} \qquad E \vdash e \Rightarrow \overbrace{\texttt{Vint}_{64}\, n}^{\text{typing}} \qquad n < \texttt{sz}_t}{E \vdash t[e] \Rightarrow E(t)[n]}\ \textsc{Read}$$

Reading into an array

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

Local environments: $E : x \rightharpoonup v$

$$\frac{\overbrace{t \in E}^{\text{typing}} \qquad E \vdash e \Rightarrow \overbrace{\texttt{Vint}_{64}\, n}^{\text{typing}} \qquad \overbrace{n < \texttt{sz}_t}^{\text{error}}}{E \vdash t[e] \Rightarrow E(t)[n]}\ \textsc{Read}$$

Error rules

$$\frac{t \in E \qquad E \vdash e \Rightarrow \texttt{Vint}_{64}\, n \qquad n < \texttt{sz}_t}{E \vdash t[e] \Rightarrow E(t)[n]}\ \textsc{Read}$$

Error rules

$$\frac{t \in E \qquad E \vdash e \Rightarrow \texttt{Vint}_{64}\, n \qquad n < \texttt{sz}_t}{E \vdash t[e] \Rightarrow E(t)[n]}\textsc{Read}$$

$$\frac{t \in E \qquad E \vdash e \Rightarrow \texttt{error}}{E \vdash t[e] \Rightarrow \texttt{error}}\textsc{ReadPathErr}$$

$$\frac{t \in E \qquad E \vdash e \Rightarrow \texttt{Vint}_{64}\, n \qquad n \geq \texttt{sz}_t}{E \vdash t[e] \Rightarrow \texttt{error}}\textsc{ReadErr}$$

Error rules

$$\frac{E \vdash e_1 \Rightarrow \texttt{Vint}\ n_1 \qquad E \vdash e_2 \Rightarrow \texttt{Vint}\ n_2 \qquad n_2 \neq 0 \qquad n_1 \neq \texttt{min\_int} \vee n_2 \neq -1}{E \vdash \texttt{divs}(e_1, e_2) \Rightarrow \texttt{Vint}\ (n_1 / n_2)}\ \textsc{Divs}$$

$$\frac{\forall j < i, E \vdash e_j \Rightarrow v_j \qquad E \vdash e_i \Rightarrow \texttt{error}}{E \vdash \texttt{divs}(e_1, e_2) \Rightarrow \texttt{error}}\ \textsc{DivsErr}_i$$

$$\frac{\dots \quad n_1 = \texttt{min\_int} \wedge n_2 = -1}{E \vdash \texttt{divs}(e_1, e_2) \Rightarrow \texttt{error}}\ \textsc{DivsErrOvf}$$

$$\frac{\dots \quad n_2 = 0}{E \vdash \texttt{divs}(e_1, e_2) \Rightarrow \texttt{error}}\ \textsc{DivsErrZero}$$

Writing into an array

$$\frac{\texttt{perm}(t) \geq \texttt{Mutable} \qquad E \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad n < \texttt{sz}_t \qquad \ldots \qquad E \vdash e' \Rightarrow v \qquad \texttt{primitive}(v)}{(E, \langle t[e] = e' \rangle, k) \to (E[t \leftarrow E(t)[n \leftarrow v]], \langle\rangle, k)} \textsc{Write}$$

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

Function call and return

Function call:

$$E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args}$$
$$\texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v})$$

$$\frac{}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \to (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function call and return

Function call:

$$E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args}$$

$$\texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v})$$

$$\forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i)$$

$$\frac{}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \to (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function call and return

Function call:

$$E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args}$$

$$\texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v})$$

$$\forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i)$$

$$\forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \wedge i \neq j \Rightarrow a_i \neq a_j$$

$$\frac{}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \rightarrow (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function call and return

Function call:

$$E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args}$$

$$\texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v})$$

$$\forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i)$$

$$\forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \wedge i \neq j \Rightarrow a_i \neq a_j$$

$$\texttt{valid\_call}(E, \texttt{f}, \vec{a})$$

$$\frac{}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \rightarrow (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function call and return

Function call:

$$\frac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args} \\ \texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v}) \\ \forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i) \\ \forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \wedge i \neq j \Rightarrow a_i \neq a_j \\ \texttt{valid\_call}(E, \texttt{f}, \vec{a}) \\ m = \{(a_i, x_i) \mid \texttt{f.perm}(x_i) = \texttt{Mutable}\} \end{array}}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \to (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function call and return

Function call:

$$\frac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \vec{v} \in \texttt{f.sig\_args} \\ \texttt{f.params} = \vec{x} \qquad E_{\texttt{f}} = \texttt{build\_env}(\vec{x}, \vec{v}) \\ \forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i) \\ \forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \wedge i \neq j \Rightarrow a_i \neq a_j \\ \texttt{valid\_call}(E, \texttt{f}, \vec{a}) \\ m = \{(a_i, x_i) \mid \texttt{f.perm}(x_i) = \texttt{Mutable}\} \end{array}}{(E, \langle y = \texttt{f}(\vec{a}) \rangle, k) \to (E_{\texttt{f}}, \texttt{f.body}, \texttt{Kreturnto}(y, E, m, k))}\ \textsc{Call}$$

Function return:

$$\frac{}{\left(E_{\texttt{f}}, \langle \texttt{return}\ v \rangle, \texttt{Kreturnto}\left(y, E, \overrightarrow{(a, x)}, k\right)\right) \to \left(E\left[\vec{a} \leftarrow \overrightarrow{E_{\texttt{f}}(x)}\right][y \leftarrow v], \langle\rangle, k\right)}\ \textsc{Return}$$

Split

$$E(t) = \texttt{Varr}\ \vec{v} \qquad E(e) = \texttt{Vint}_{64} i \qquad E(\texttt{sz}_t) = \texttt{Vint}_{64} n \qquad i \leq n$$

$$\ldots$$

$$o_j = \left\{ x \mid \texttt{sz}_{u_j} = \texttt{sz}_x \right\}$$

$$E' = (E \setminus (o_1 \cup o_2 \cup \{t\})) \begin{bmatrix} \texttt{sz}_{u_1} \leftarrow \texttt{Vint}_{64} i, & \texttt{sz}_{u_2} \leftarrow \texttt{Vint}_{64}(n-i), \\ u_1 \leftarrow \texttt{Varr}\ \left(\vec{v}_{0..(i-1)}\right), & u_2 \leftarrow \texttt{Varr}\ \left(\vec{v}_{i..(n-1)}\right) \end{bmatrix}$$

$$\frac{}{(E, \langle u_1, u_2 \leftarrow \texttt{split}(t, e)\{s\} \rangle, k) \rightarrow (E', \langle s \rangle, \texttt{Kjoin}(t, u_1, u_2, k))}$$

$$E(u_1) = \texttt{Varr}\ \vec{v} \qquad E(u_2) = \texttt{Varr}\ \vec{w} \qquad \ldots$$

$$\frac{}{(E, \langle\rangle, \texttt{Kjoin}(t, u_1, u_2, k)) \rightarrow ((E \setminus \{u_1, u_2\})[t \leftarrow \texttt{Varr}\ (v +\!\!+ w)], \langle\rangle, k)}$$

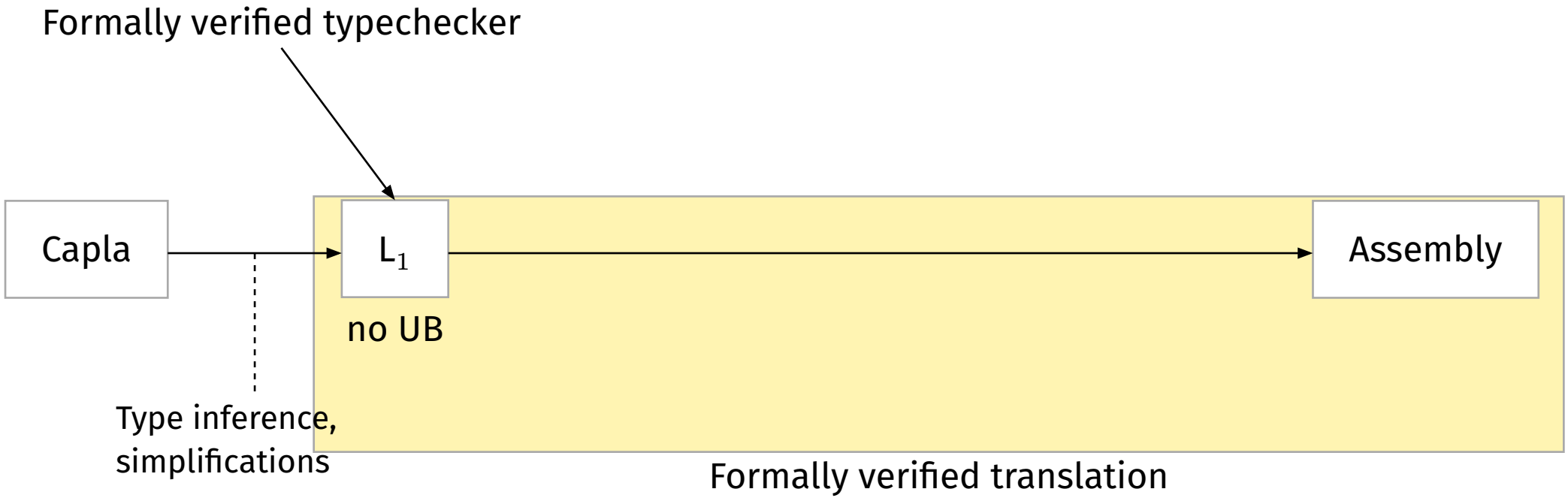
Dynamic allocation

$$\frac{\texttt{perm}(t) = \texttt{Owned} \qquad E \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad \ldots \qquad t \notin k \qquad o = \{x \mid \texttt{sz}_t = \texttt{sz}_x\}}{(E, \langle t \leftarrow \texttt{alloc}(\texttt{i32}, e)\rangle, k) \to ((E \setminus o)[t \leftarrow \texttt{Varr}\ [0, 0, \ldots]][\texttt{sz}_t \leftarrow \texttt{Vint}_{64} n], \langle\rangle, k)}\ \textsc{Alloc}$$

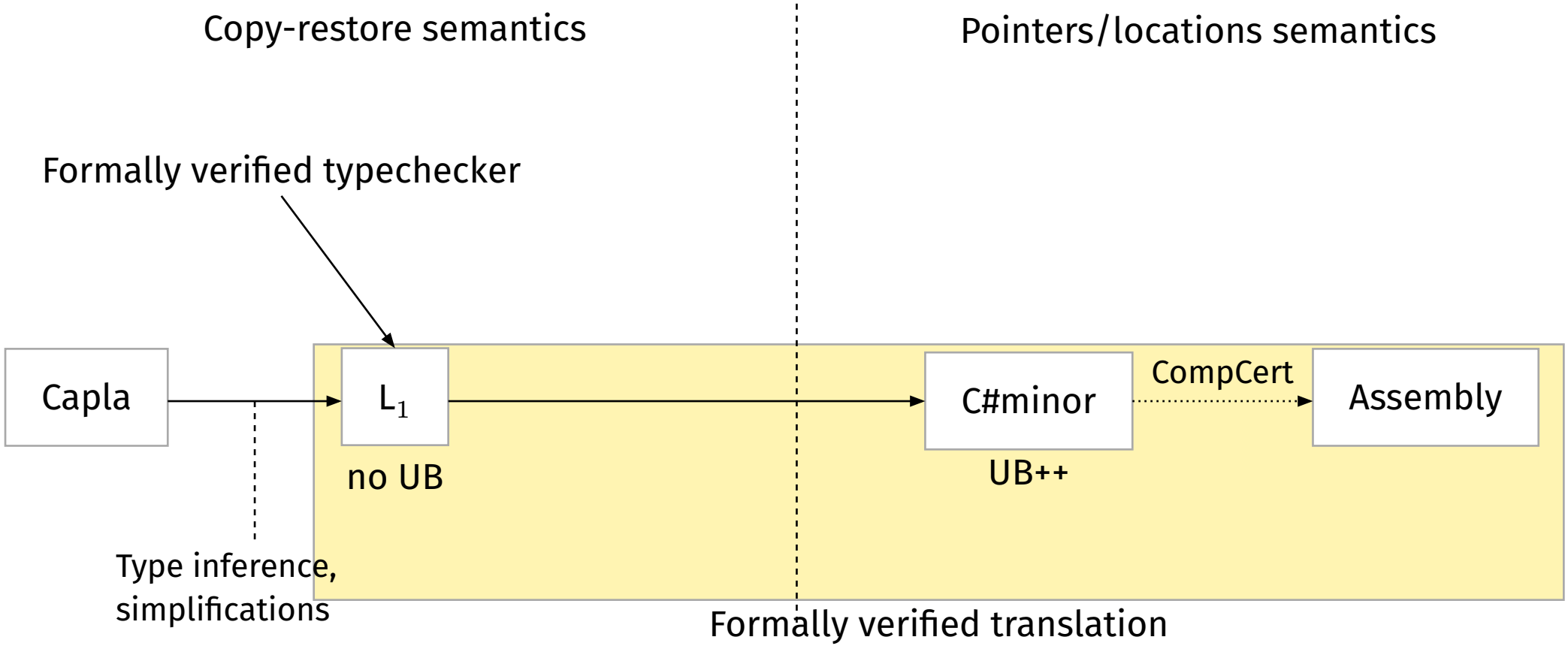
$$\frac{\texttt{perm}(t) = \texttt{Owned} \qquad \ldots \qquad t \notin k}{(E, \langle \texttt{free}\ t\rangle, k) \to (E \setminus \{t, \texttt{sz}_t\}, \langle\rangle, k)}\ \textsc{Free}$$

Compiler architecture and benchmarks

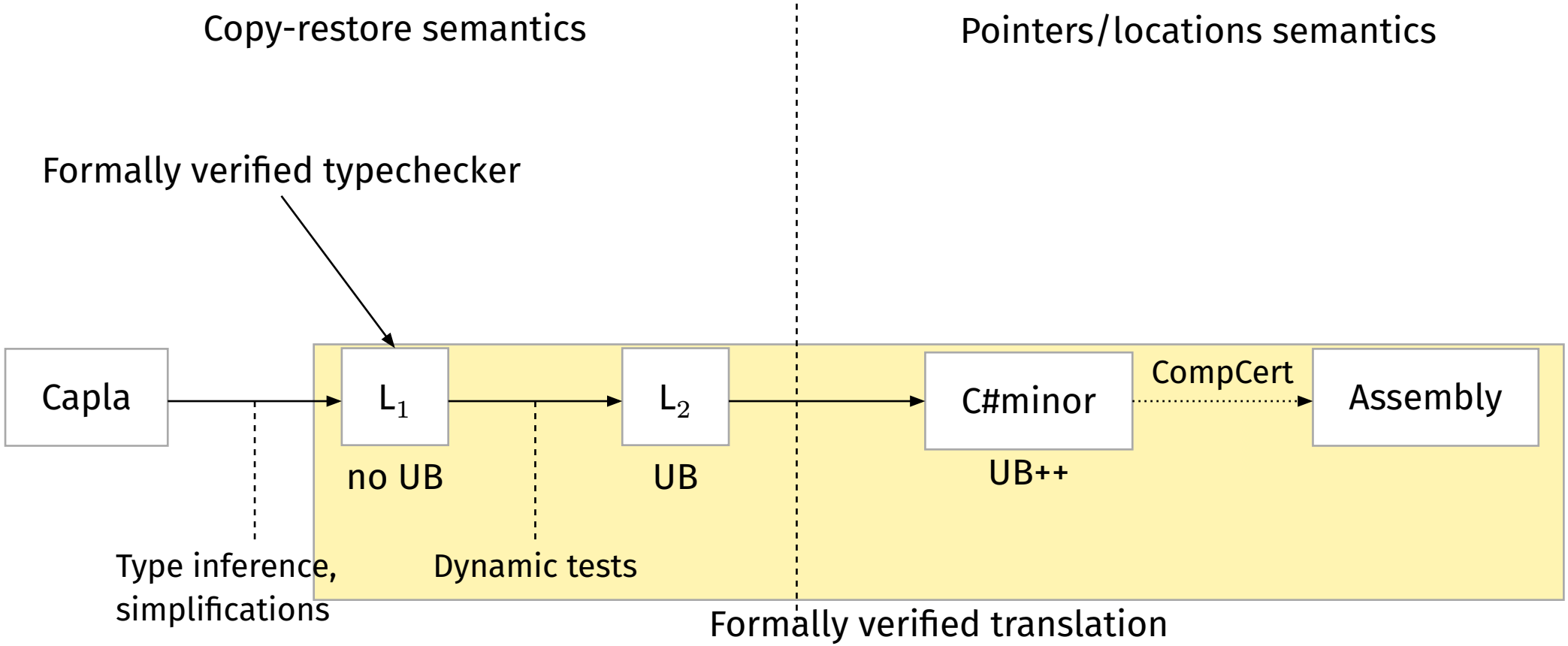
Compiler architecture



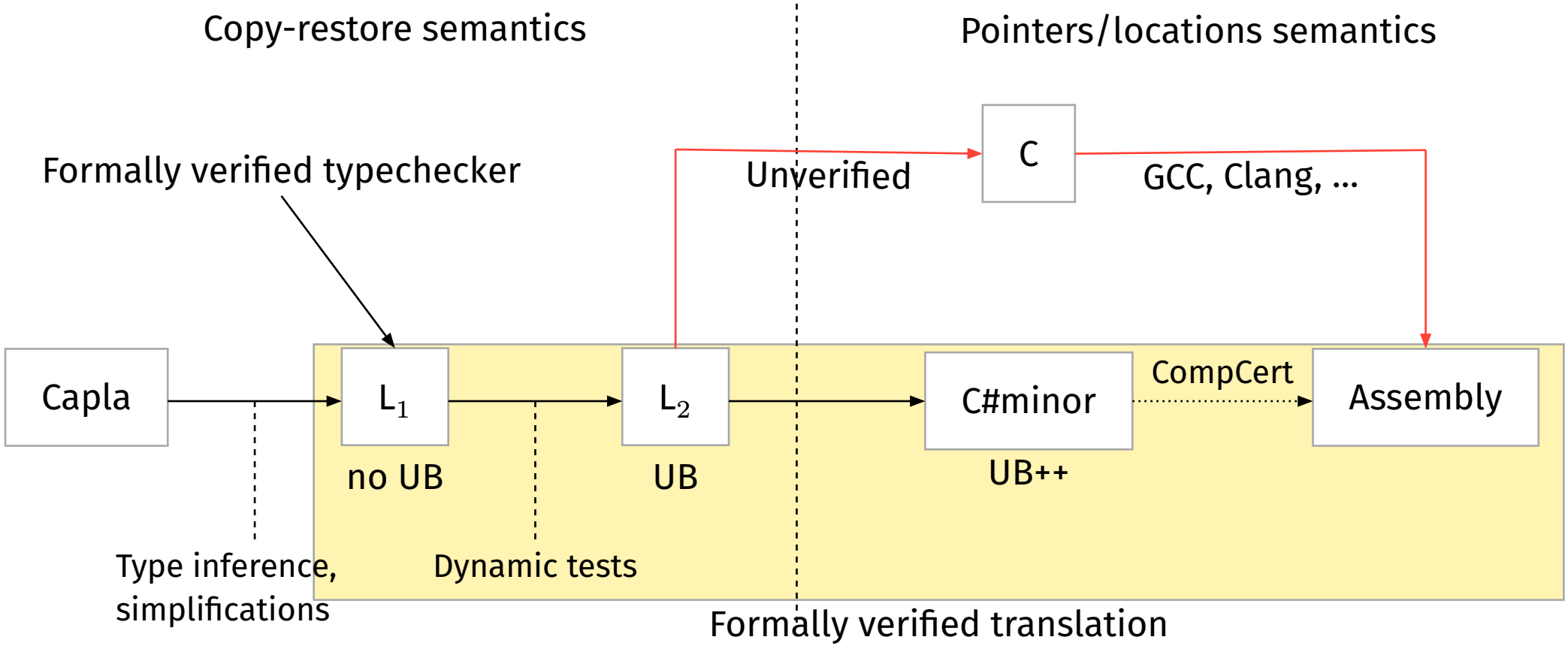
Compiler architecture



Compiler architecture



Compiler architecture



Benchmarks

	CompCert
`zdotu`	2.25
`saxpy`	5.37
`sgemv`	2.86
`dgemv`	1.73
`dtrsv` (N)	2.87
`dtrsv` (T)	2.30
`mpn_addmul_1`	2.74
`mpn_mul`	3.62
`mpn_mul` (with assembly)	1.59

- ≥ 1: slower than the original implementation
- Input vectors/matrices are small enough to minimize cache misses
- Reference x86-64 BLAS/LAPACK 3.12.0 (Fortran) and GMP 6.3.0 (handwritten assembly)
- LLVM 19.1.7 and GCC 14.2.1, optimization level `-O2 -ftree-vectorize`

Benchmarks

	CompCert	GCC	LLVM
`zdotu`	2.25	1.01	0.93
`saxpy`	5.37	4.25	0.78
`sgemv`	2.86	1.32	0.96
`dgemv`	1.73	1.03	0.59
`dtrsv` (N)	2.87	1.32	1.78
`dtrsv` (T)	2.30	1.14	1.35
`mpn_addmul_1`	2.74	1.19	1.08
`mpn_mul`	3.62	1.93	1.80
`mpn_mul` (with assembly)	1.59	1.42	1.42

- ≥ 1: slower than the original implementation
- Input vectors/matrices are small enough to minimize cache misses
- Reference x86-64 BLAS/LAPACK 3.12.0 (Fortran) and GMP 6.3.0 (handwritten assembly)
- LLVM 19.1.7 and GCC 14.2.1, optimization level `-O2 -ftree-vectorize`

Benchmarks

	CompCert	GCC	LLVM	GCC with assertions	LLVM with 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	1.19	1.08	–	–
`mpn_mul`	3.62	1.93	1.80	–	–
`mpn_mul` (with assembly)	1.59	1.42	1.42	–	–

- ≥ 1: slower than the original implementation
- Input vectors/matrices are small enough to minimize cache misses
- Reference x86-64 BLAS/LAPACK 3.12.0 (Fortran) and GMP 6.3.0 (handwritten assembly)
- LLVM 19.1.7 and GCC 14.2.1, optimization level `-O2 -ftree-vectorize`

Type safety

Typing

Main difficulty of typing: tracking initialized variables.

Typing judgments

$$\{V\}\ c\ \{V'\}$$

Meaning:

The execution of c requires the variables of V to be initialized and ensures that the variables of V' are initialized.

- Backward typing: typing from the end of the function to the beginning.

Assignment

$$\frac{\Gamma \vdash e : \tau, V_e \qquad \Gamma(x) = \tau}{\{V \setminus \{x\} \cup V_e\}\ x = e\ \{V\}}$$

$$\frac{\Gamma \vdash e' : \tau, V_{e'} \qquad \Gamma(x) = \texttt{array}\ \tau \qquad \Gamma \vdash e : \texttt{int}_{64}, V_e}{\{V \cup V_{e'} \cup V_e \cup \{x\}\}\ x[e] = e'\ \{V\}}$$

Dynamic allocation

$$\frac{\Gamma \vdash e : \texttt{int}_{64}, V_e}{\{V \setminus \{t, \texttt{sz}_t\} \cup V_e\}\ t \leftarrow \texttt{alloc}(\texttt{i32}, e)\ \{V\}}$$

$$\frac{\{t, \texttt{sz}_t\} \cap V = \emptyset}{\{V \cup \{t, \texttt{sz}_t\}\}\ \ \texttt{free}\ t\ \{V\}}$$

Blocks

$$\frac{\{V'\}\ s\ \{(V, V :: X)\}}{\{V'\}\ \{s\}\ \{(V, X)\}} \qquad \frac{}{\{X[n]\}\ \texttt{exit}\ n\ \{(V, X)\}}$$

Sequence and loop

$$\frac{\{V_1\}\ s_1\ \{(V_2, X)\} \qquad \{V_2\}\ s_2\ \{(V, X)\}}{\{V_1\}\ s_1; s_2\ \{(V, X)\}}$$

$$\frac{\overbrace{\{V''\}\ s\ \{(V', X)\}}^{\text{avoid double free}} \qquad \{V'\}\ s\ \{(V, X)\}}{\{V''\}\ \texttt{loop}\ s\ \{(V, X)\}}$$

Type safety

Invariants

1. The content of the local environment has the expected type.
2. The needed variables are initialized.
3. Array sizes are valid (*i.e.*, at most the actual size of array values).

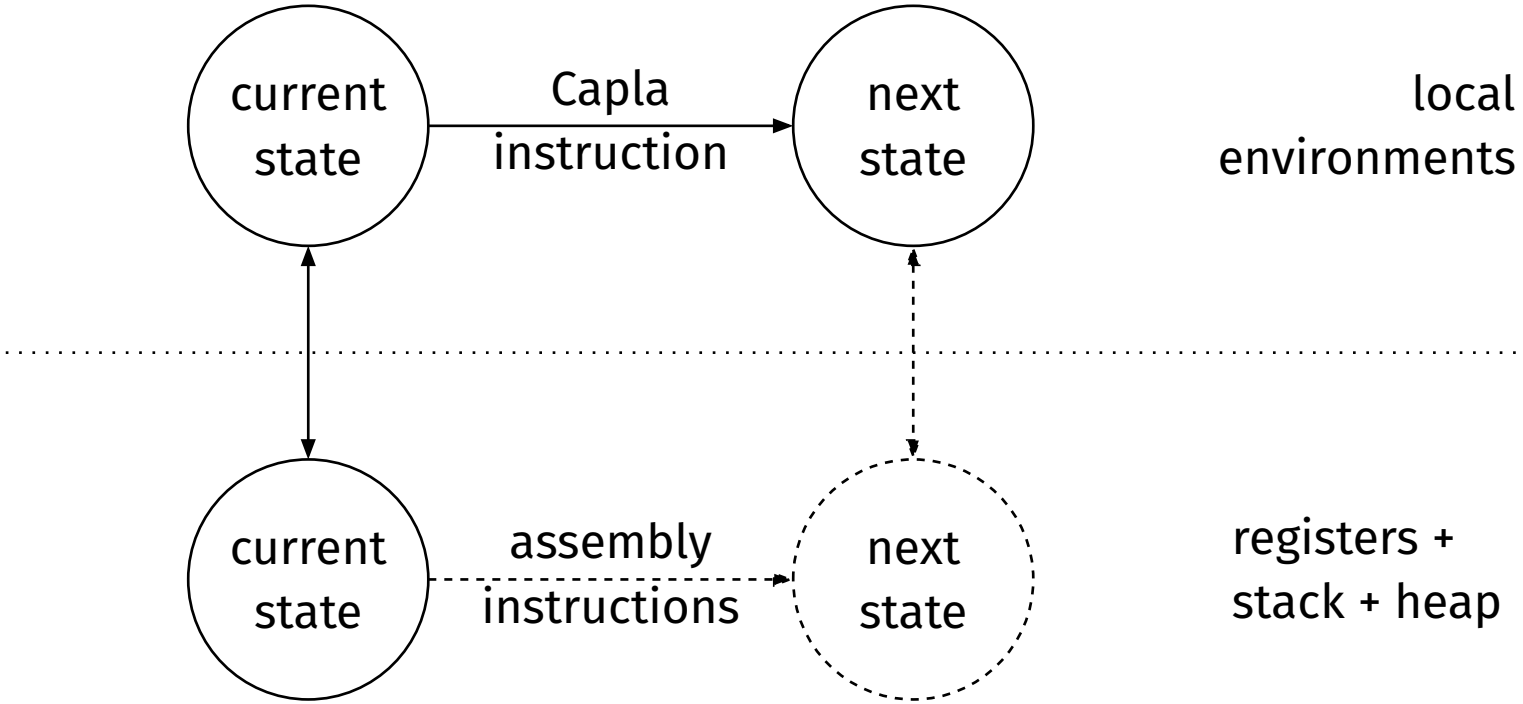
Theorem (Type safety)

Given a successfully typed L_1 program,
if all the invariants hold in a non-final L_1 state s,
then there exists a state t such that $s \to t$ and all the invariants hold in t.

Semantics preservation

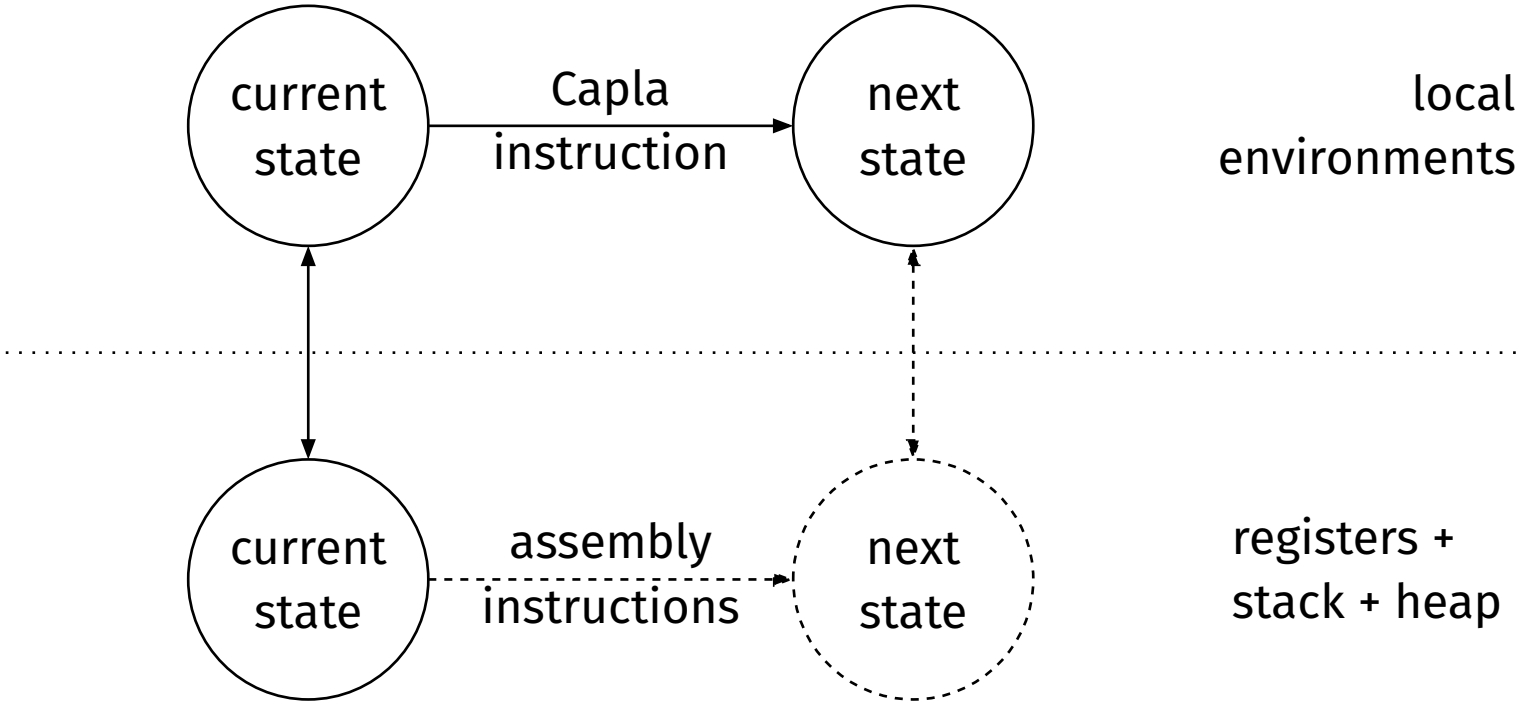
Semantics preservation

Forward simulation: each step in Capla implies a sequence of no-UB steps in assembly and execution states are still related



Semantics preservation

Forward simulation: each step in Capla implies a sequence of no-UB steps in assembly and execution states are still related



Cminor / ASM is deterministic, so forward simulation implies backward simulation.

Semantics preservation from L_2 to C#minor: separation

Memory element:

$$\texttt{Mem} \equiv \texttt{option}\big(\texttt{block} \times \texttt{offset} \times \texttt{element size} \times \texttt{list}\big(\texttt{Mem}\big)\big)$$

Semantics preservation from L_2 to C#minor: separation

Memory element:

$$\texttt{Mem} \equiv \texttt{option}\big(\texttt{block} \times \texttt{offset} \times \texttt{element size} \times \texttt{list}\big(\texttt{Mem}\big)\big)$$

$$\begin{aligned} \texttt{Vint}\ n &\to \texttt{None} \\ \texttt{Varr}\ \vec{v} &\to \texttt{Some}\ \left(b, o, s, \vec{l}\right) \end{aligned}$$

Semantics preservation from L_2 to C#minor: separation

Memory element:

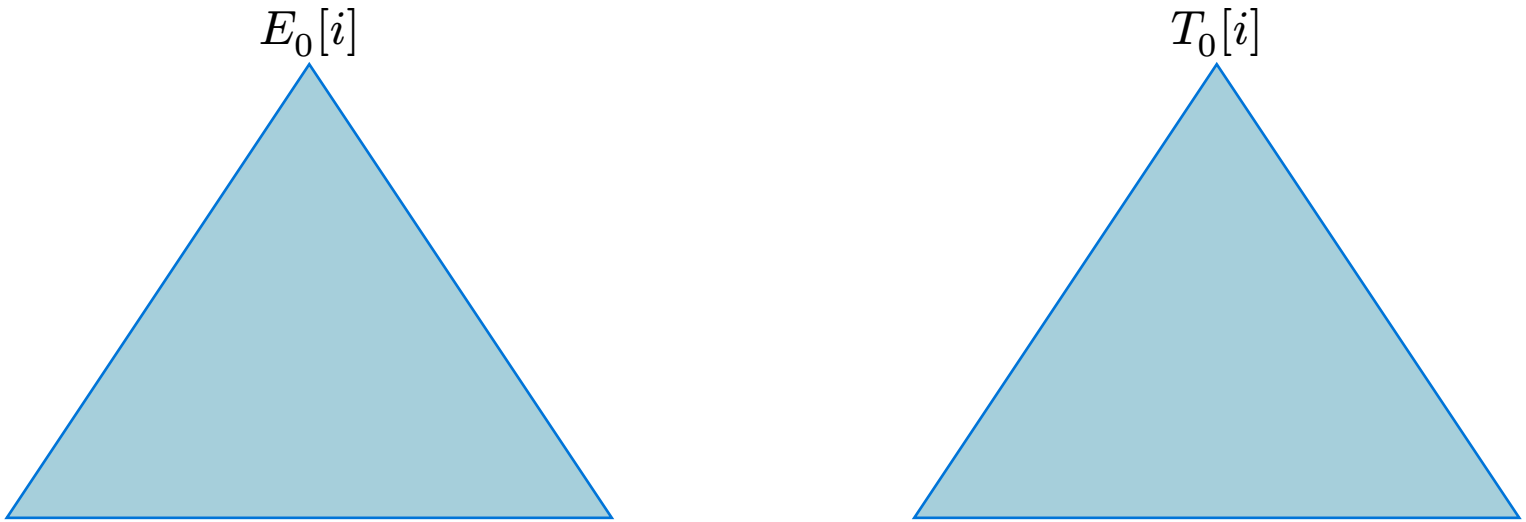
$$\texttt{Mem} \equiv \texttt{option}\big(\texttt{block} \times \texttt{offset} \times \texttt{element size} \times \texttt{list}\big(\texttt{Mem}\big)\big)$$

$$\begin{aligned} \texttt{Vint}\ n &\rightarrow \texttt{None} \\ \texttt{Varr}\ \vec{v} &\rightarrow \texttt{Some}\ \left(b, o, s, \vec{l}\right) \end{aligned}$$

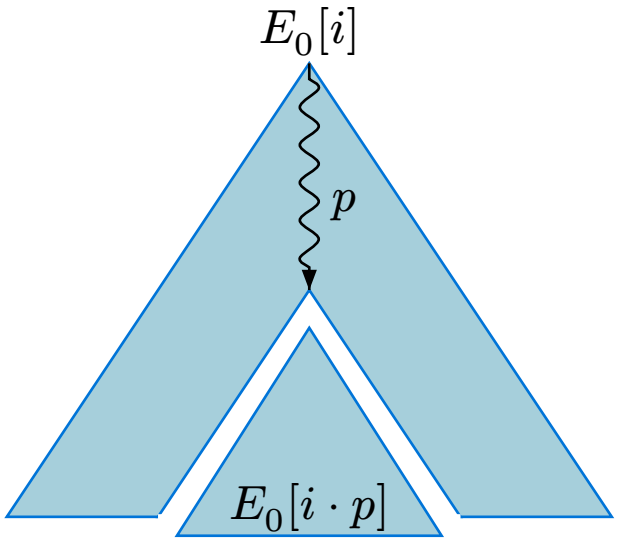
Translation function:

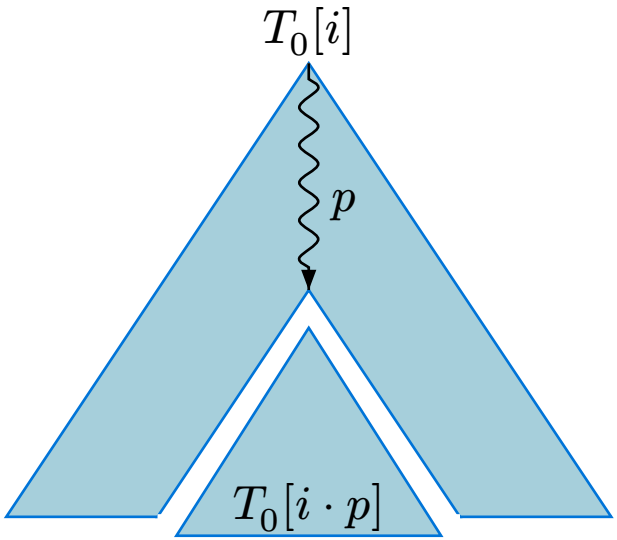
$$T : \texttt{path} \rightarrow \texttt{Mem}$$

Semantics preservation from L_2 to C#minor: separation

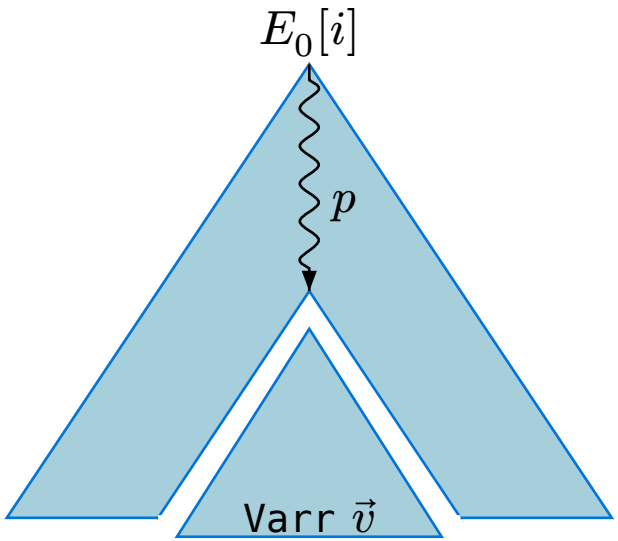


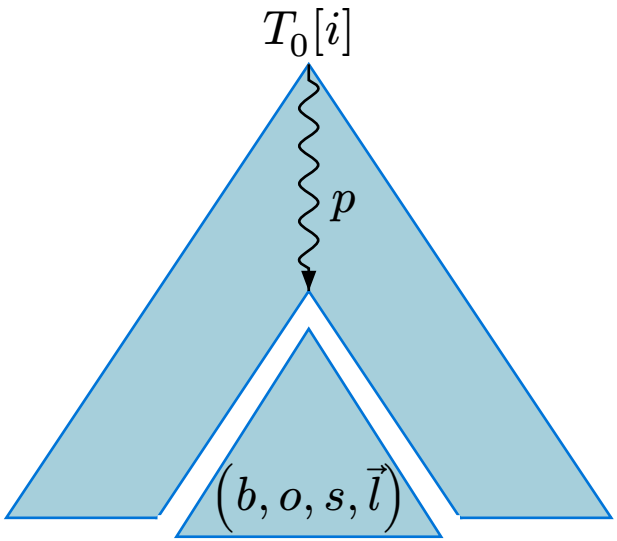
Semantics preservation from L_2 to C#minor: separation



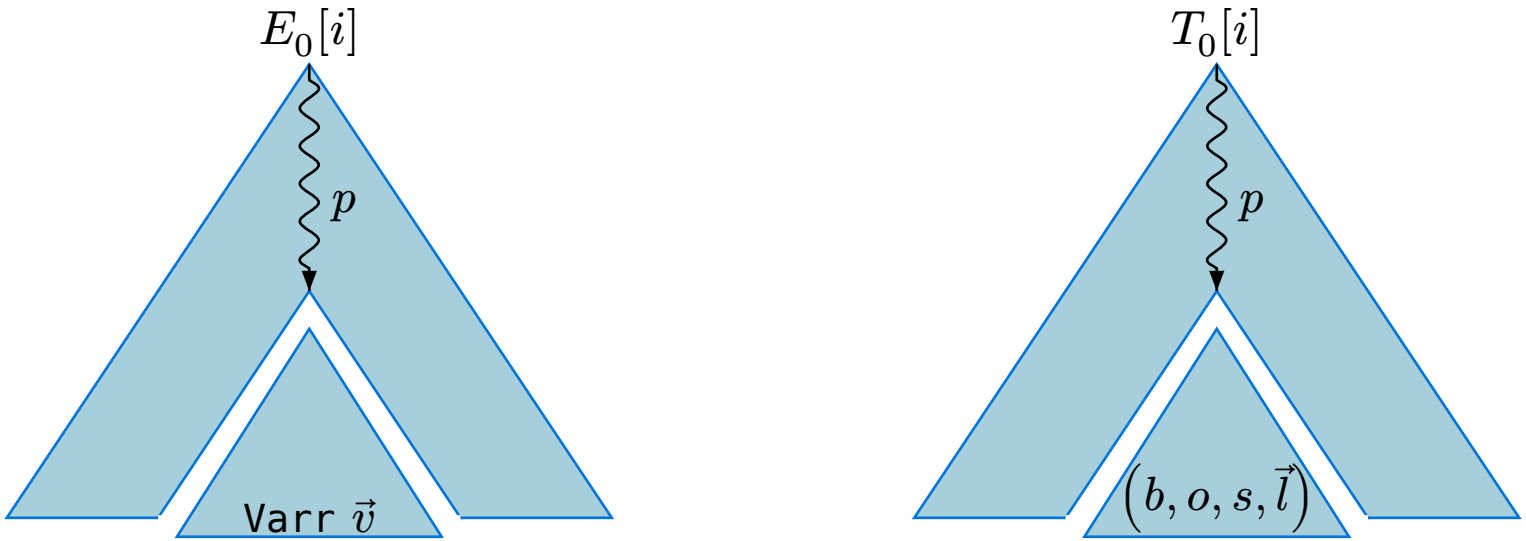


Semantics preservation from L_2 to C#minor: separation



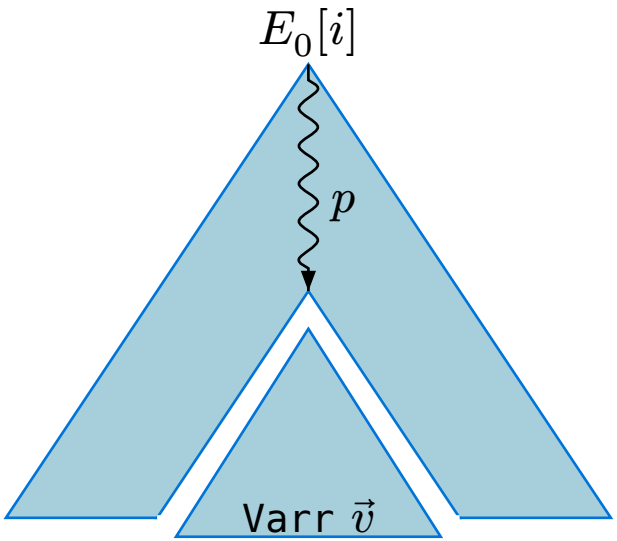


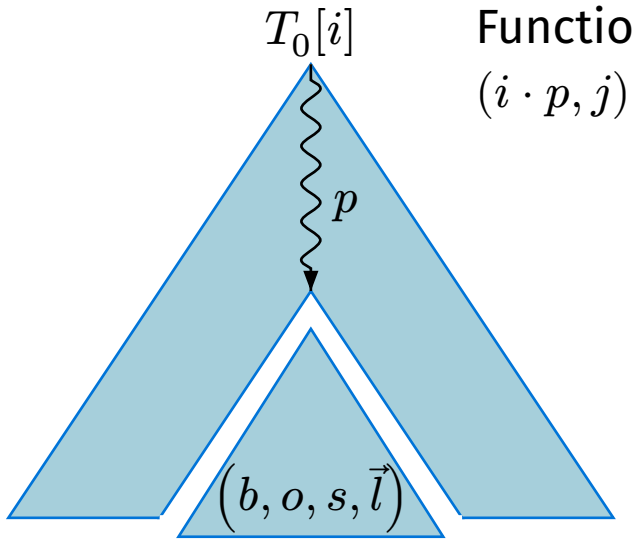
Semantics preservation from L_2 to C#minor: separation



$$\forall k, k < |\vec{v}| \Rightarrow M[b, o + k \cdot s] \sim v_k$$

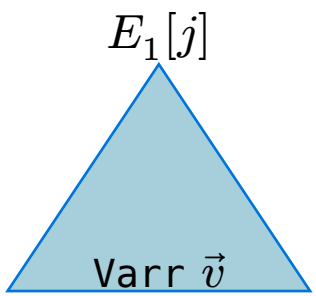
Semantics preservation from L_2 to C#minor: separation

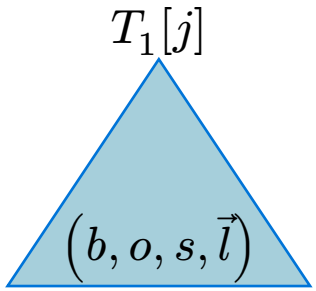




Function call

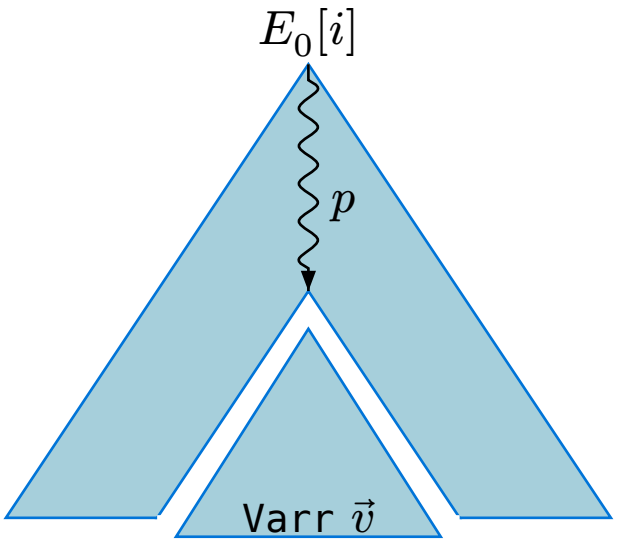
$(i \cdot p, j) \in m$

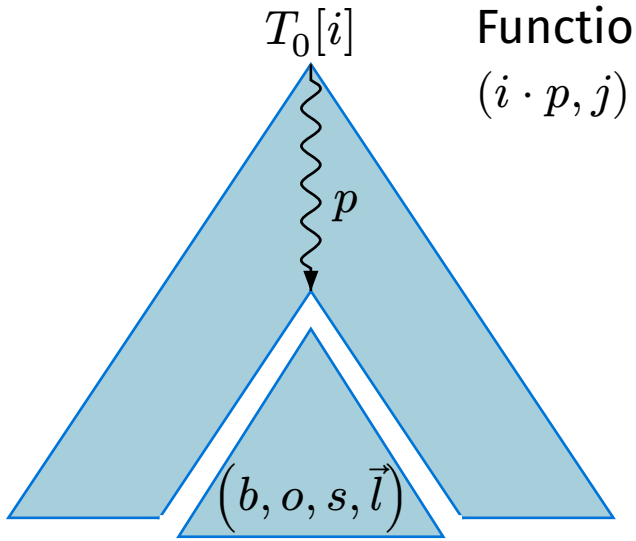




$$\forall k, k < |\vec{v}| \Rightarrow M[b, o + k \cdot s] \sim v_k$$

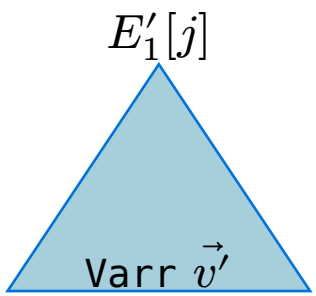
Semantics preservation from L_2 to C#minor: separation

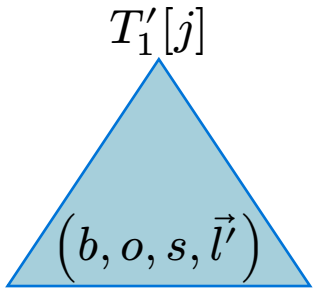




Function call

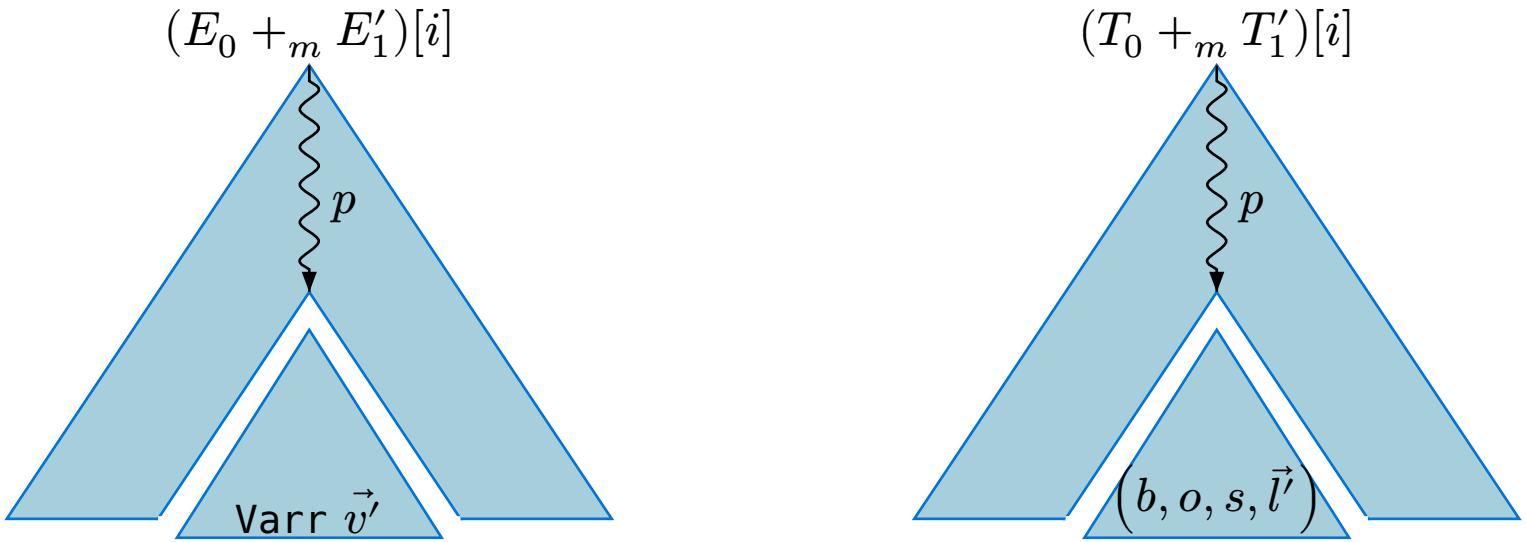
$(i \cdot p, j) \in m$





$$\forall k, k < |\vec{v'}| \Rightarrow M'[b, o + k \cdot s] \sim v'_k$$

Semantics preservation from L_2 to C#minor: separation



$$\forall k, k < |\vec{v'}| \Rightarrow M'[b, o + k \cdot s] \sim v'_k$$

Semantics preservation from L_2 to C#minor: separation

$$
\begin{aligned}
\texttt{separated\_mem}_p(\texttt{None}, \_) &= \texttt{true} \\
\texttt{separated\_mem}_p(\_, \texttt{None}) &= \texttt{true} \\
\texttt{separated\_mem}_{\texttt{Shared}} &= \texttt{true} \\
\texttt{separated\_mem}_{\texttt{Mutable}}((b_1, o_1, e_1, l_1), (b_2, o_2, e_2, l_2)) &= b_1 \neq b_2 \\
&\quad \vee o_1 + e_1 \cdot |l_1| \leq o_2 \\
&\quad \vee o_2 + e_2 \cdot |l_2| \leq o_1 \\
\texttt{separated\_mem}_{\texttt{Owned}}((b_1, o_1, e_1, l_1), (b_2, o_2, e_2, l_2)) &= b_1 \neq b_2
\end{aligned}
$$

$$
\texttt{separated}(T) = \forall i\ p\ i'\ p', i \cdot p \neq i' \cdot p' \Rightarrow \texttt{separated\_mem}_{\texttt{perm}(i)}(\mathrm{tr}(i \cdot p), \mathrm{tr}(i' \cdot p'))
$$

Semantics preservation

Theorem (Forward simulation)

Let $s \underset{L_2}{\to} t$, *then for every state* s' *s.t.* $s \Leftrightarrow s'$, *either*

- $|t| < |s|$, *or*
- *there exists* t' *s.t.* $s' \underset{\text{C\#minor}}{\to^*} t'$ *and* $t \Leftrightarrow t'$.

Theorem (Compiler correction - backward simulation)

Let $s \underset{\text{ASM}}{\to} t$, *then for every state* s' *s.t.* $s' \Leftrightarrow s$, *either*

- $|t| < |s|$, *or*
- *there exists* t' *s.t.* $s' \underset{L_1}{\to^+} t'$ *and* $t' \Leftrightarrow t$.

Conclusion

Conclusion

- Safe language for computer algebra
  - Suitable for array-based programs
  - Non-aliasing policy for mutable memory
  - Simple semantics for easier program reasoning

```
// 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)
```

- Correct compiler to assembly
  - Correctness and safety (WIP split) proofs are verified with Rocq
    ∼10,000 lines of code, ∼11,000 lines of spec, ∼17,000 lines of proof
- Unverified C backend for performance

Work in progress - Proof of programs

- Big-step semantics with proof of soundness
- Weakest Precondition computation (work by Oliver Turner) with proof of soundness
- Used on small proofs

Future work:

- Prove the soundness of the weakest precondition computation with the small-step semantics
- Improve usability while proving a function

Conclusion

This work is part of the ERC Fresco project: https://fresco.gitlabpages.inria.fr/capla/language/index.html

- Goal: Turn Rocq into a computer algebra system

```
fun mpn_addmul_1(rp: mut [u64; n],
                 up: [u64; n],
                 n v0: u64) -> u64 {
  assert (n >= 1);
  let u0 crec c p1 p0 r0: u64; crec = 0;
  for i : u64 = 0 .. n {
    u0 = up[i];
    p0 = u0 * v0;
    p1 = __builtin_umulh64(u0, v0);
    r0 = rp[i];
    p0 = r0 + p0;
    c = (u64) (r0 > p0);
    p1 = p1 + c;
    r0 = p0 + crec;
    c = (u64) (p0 > r0);
    crec = p1 + c;
    rp[i] = r0;
  }
  return crec; }
```

Appendix

$$(E, s, k) \leftrightarrow \left(M, \hat{E}, \texttt{comp}(s), \hat{k}\right) \equiv \exists \vec{T}, \begin{cases} \texttt{separated}(T) \\ E \overset{T_0}{\longleftrightarrow} \left(\hat{E}, M\right) \\ (E, k) \overset{\vec{T}}{\leftrightarrow} \left(M, \hat{k}\right) \end{cases}$$

$$E \overset{T}{\leftrightarrow} \left(\hat{E}, M\right) \equiv \forall (i) E[i] \sim \hat{E}[i] \wedge \forall (p) E[p] \overset{T}{\sim} M$$

$$(E, \texttt{Kseq}(s, k)) \overset{\vec{T}}{\leftrightarrow} \left(M, \texttt{Kseq}\left(\texttt{comp}(s), \hat{k}\right)\right) \equiv (E, k) \overset{\vec{T}}{\leftrightarrow} \left(M, \hat{k}\right)$$

$$(E_1, \texttt{Kreturnto}(E_0, m, k)) \overset{T_1 :: T_0 :: \vec{\mathrm{T}}}{\longleftrightarrow} \left(M, \texttt{Kreturnto}\left(\widehat{E_0}, \hat{k}\right)\right) \equiv \begin{cases} \texttt{separated}(T_0 +_m T) \\ (E_0 +_m E_1) \overset{T_0 +_m T}{\longleftrightarrow} \left(\widehat{E_0}, M\right) \\ (E_0 +_m E_1, k) \overset{T_0 +_m T :: \vec{T}}{\longleftrightarrow} \left(M, \hat{k}\right) \end{cases}$$

Proof in Coq

	Code	Specification	Proof
Parser and type inference for L_0 (OCaml)	6972	–	–
Syntax and types of L_1 and L_2	1270	–	404
Common semantics definitions and proofs	–	1975	1734
Semantics of L_1	–	1191	734
Semantics of L_2	–	469	96
Type checking and type safety of L_1	927	1590	3267
Compilation from L_1 to L_2	456	1486	3992
Compilation from L_2 to C#minor	513	2793	6169
Miscellaneous	–	1428	1275
Total	10138	10932	17671