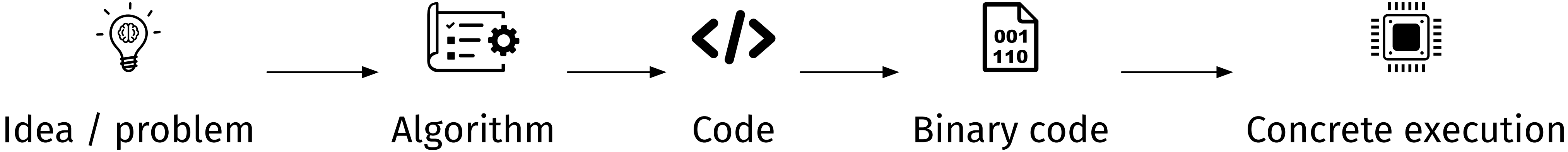
Programming language and formally verified compiler for low-level numerical libraries

Josué Moreau

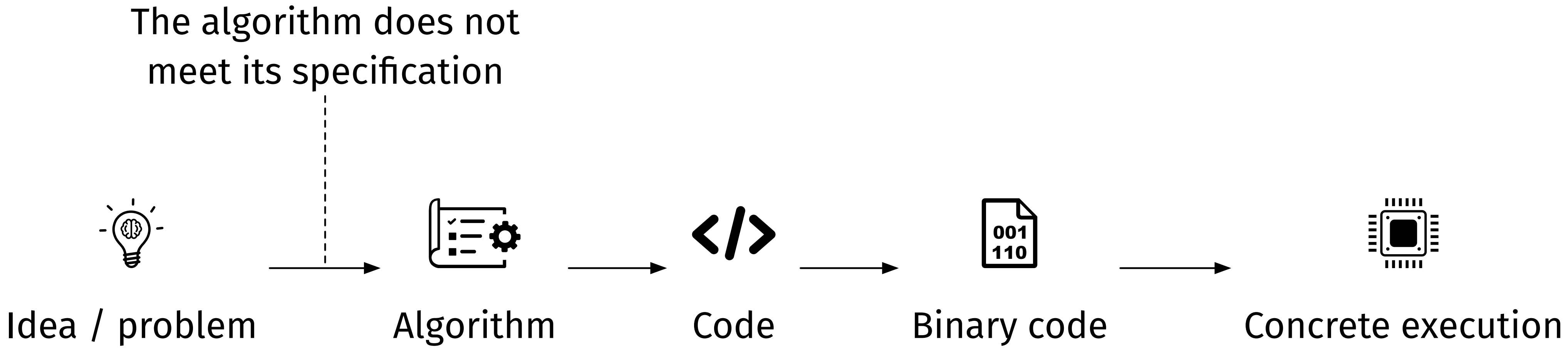
December 18, 2025

Bugs in the steps of programming



Idea / problem → Algorithm → Code → Binary code → Concrete execution

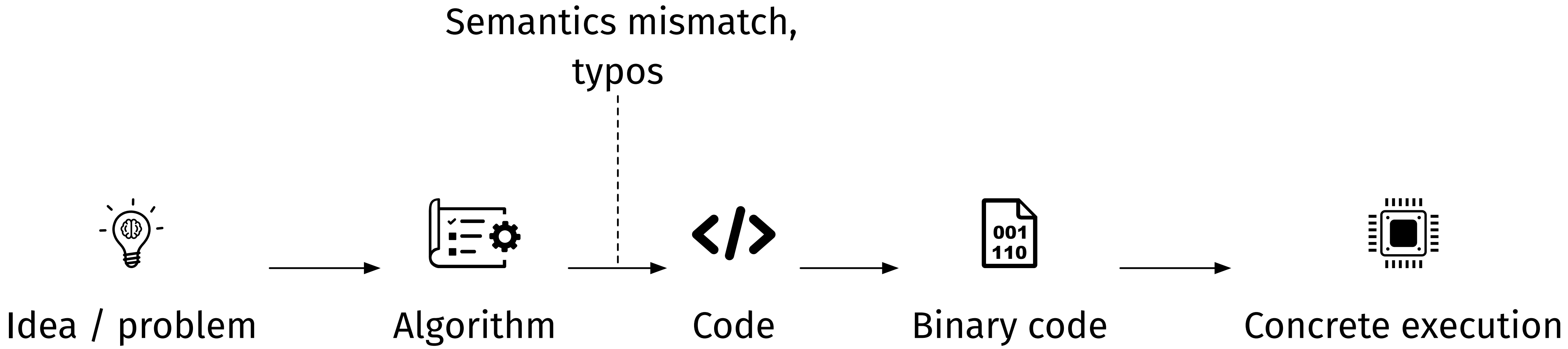
Bugs in the steps of programming



The algorithm does not meet its specification

Idea / problem → Algorithm → Code → Binary code → Concrete execution

Bugs in the steps of programming



Semantics mismatch, typos

Idea / problem

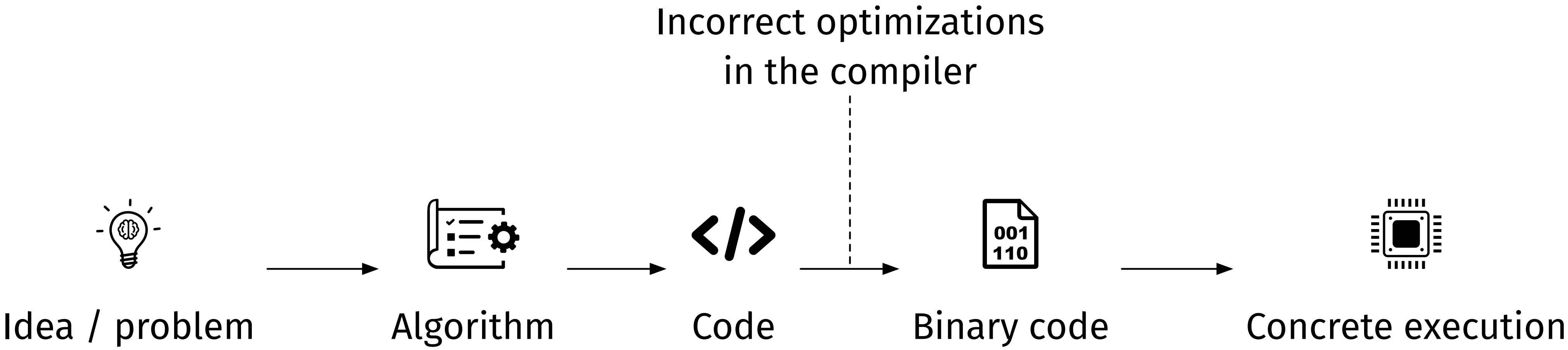
Algorithm

Code

Binary code

Concrete execution

Bugs in the steps of programming



Incorrect optimizations in the compiler

Idea / problem

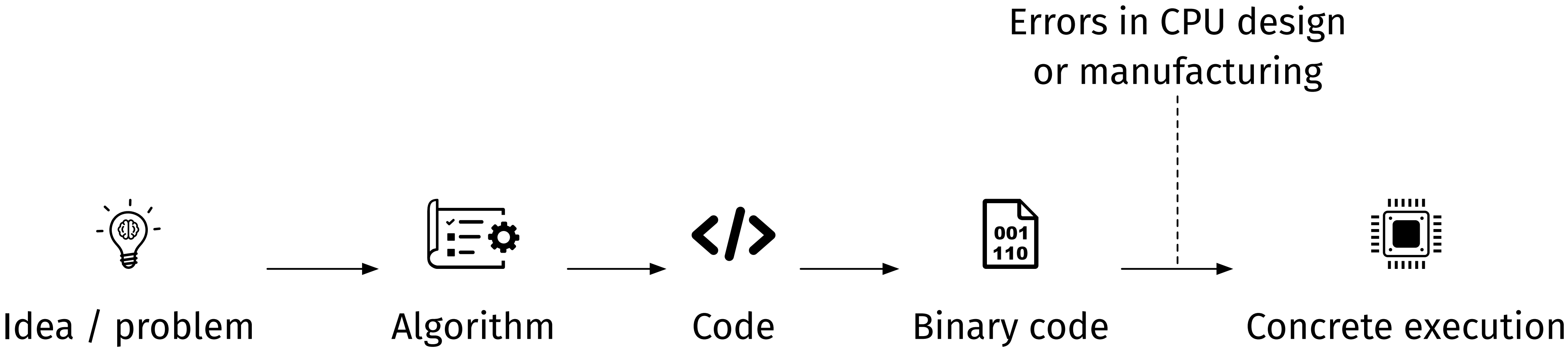
Algorithm

Code

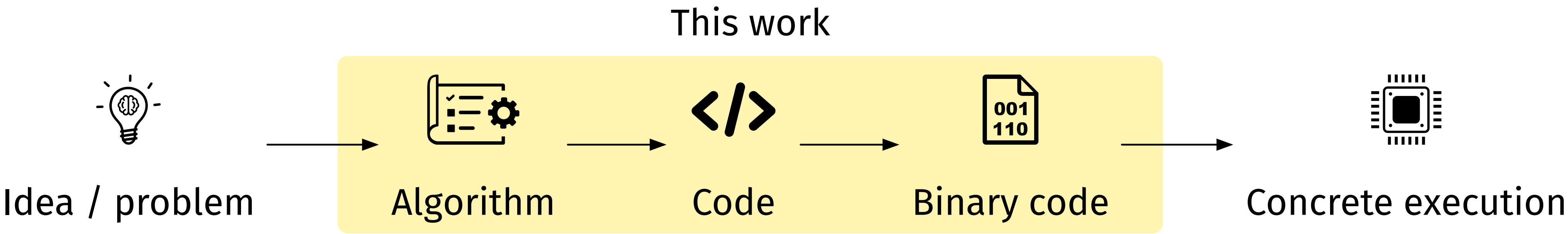
Binary code

Concrete execution

Bugs in the steps of programming



Bugs in the steps of programming



Low-level numerical libraries

Designed and optimized to work with a specific **concrete** mathematical object

Examples of widely used libraries:

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

Reliability of low-level numerical libraries

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 decimal digits or `m` is at least 7000 decimal digits.

Reliability of low-level numerical libraries

Bug (GMP $\leq$ 5.1.1)

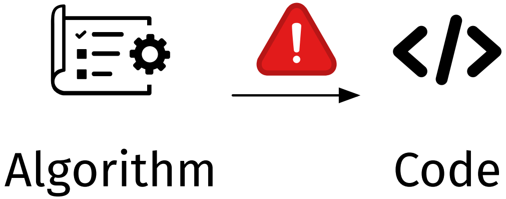
`mpz_pown_ui(r, b, e, m)` : $r \leftarrow b^e \bmod m$

Computes garbage if `b` is over 15000 decimal digits or `m` is at least 7000 decimal digits.

```
+      /* We need to allocate separate remainder area, since mpn_mu_div_qr does
+        not handle overlap between the numerator and remainder areas.
+        FIXME: Make it handle such overlap.  */
+      mp_ptr rp = TMP_ALLOC_LIMBS (dn);
      mp_size_t itch = mpn_mu_div_qr_itch (nn, dn, 0);
      mp_ptr scratch = TMP_ALLOC_LIMBS (itch);
-      mpn_mu_div_qr (qp, np, np, nn, dp, dn, scratch);
+      mpn_mu_div_qr (qp, rp, np, nn, dp, dn, scratch);
+      MPN_COPY (np, rp, dn);
```

Output

Input

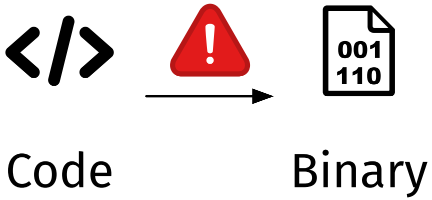


Algorithm

Code

— Commit `17d8eb27d0f4`

Compiler can have bugs



> MacOS Xcode 11 prior to 11.3 miscompiles GMP 6.2.0,
> leading to crashes and miscomputation.
>
> — GMP website

> – Clearly, GMP is not compiled correctly by newer Apple compilers.
> – “Apple compilers” are actually based on LLVM clang.
> – Indeed. Plus a bunch of bugs.
>
> — GMP Bug mailing-list, “problems on MacOS 14.5 (23F79) (XCode 15.4)”

> Enable link-time optimizations in GMP 6.3.0
> Some x86 64-bit builds fail for the *mpq* tests `t-cmp`, `t-cmp_ui`, `t-cmp_z`;
> this is caused by a GCC bug where some additive algebra goes very wrong.
>
> — GMP website

The case of low-level numerical libraries

Written in C, Fortran, and assembly

Pros:

- Allow a precise management of the data representation and memory
- Compilers generate highly optimized code

Cons:

- Error-prone languages
- Difficult program verification
- Compilers are too complicated for a total confidence in the generated code

Capla

- A language dedicated to low-level numerical libraries
  - safe (no undefined behaviors)
  - a pointer-free semantics to simplify program reasoning

Capla

- A language dedicated to low-level numerical libraries
  - safe (no undefined behaviors)
  - a pointer-free semantics to simplify program reasoning

- A compiler
  - efficient generated code
  - using CompCert as backend

Capla

- A language dedicated to low-level numerical libraries
  - safe (no undefined behaviors)
  - a pointer-free semantics to simplify program reasoning
- A compiler
  - efficient generated code
  - using CompCert as backend
- Rocq formalization
  - semantics of Capla
  - compiler correctness
  - type safety

Outline

- The Capla language
- Copy-restore semantics
- Typing and safety
- Compiler and semantics preservation
- Expressiveness and benchmarks
- Conclusion

The Capla language

A first example: Pascal’s triangle

```
fun binomial(n k: u64) -> u64 {
  let t = alloc u64, k + 1; // allocate a sufficiently long line
  t[0] = 1;
  for i = 0 .. n {
    next_line(t, k + 1); // compute the i+1-th line using the i-th line
  }
  let r = t[k];
  free t;
  return r;
}

fun next_line(t: mut [u64; k], k: u64) {
  for decr i = (k - 1) .. 0 {
    t[i] = t[i] + t[i - 1];
  }
}
```

Arrays with explicit sizes are the main data structure

Focus on some Capla features

- Arrays have explicit sizes and functions have meaningful signatures

Focus on some Capla features

- Arrays have explicit sizes and functions have meaningful signatures
- Undefined behaviors are replaced by runtime errors

Focus on some Capla features

- Arrays have explicit sizes and functions have meaningful signatures
- Undefined behaviors are replaced by runtime errors
- “Views” constructs provide safe pointer arithmetic

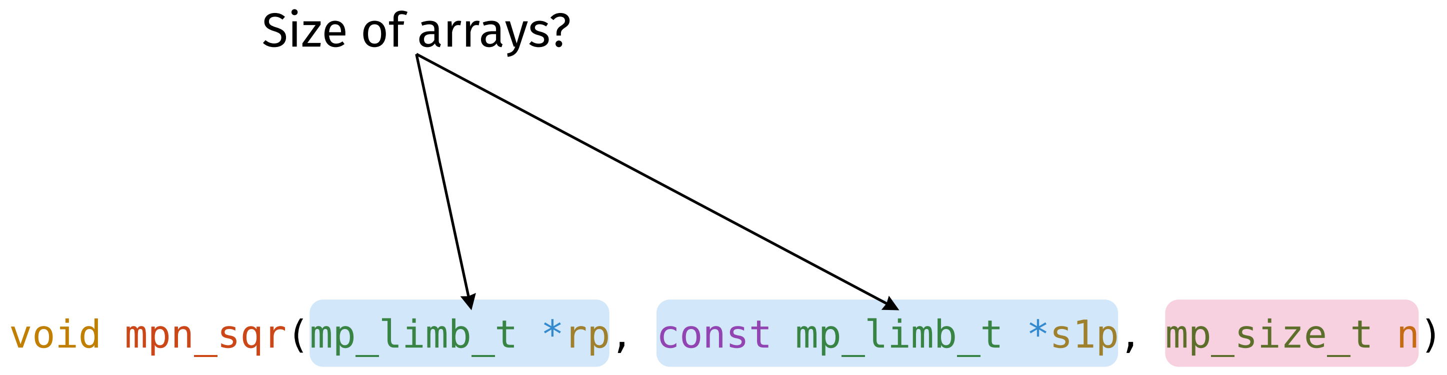
Lack of expressiveness in C signatures

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

Lack of expressiveness in C signatures

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

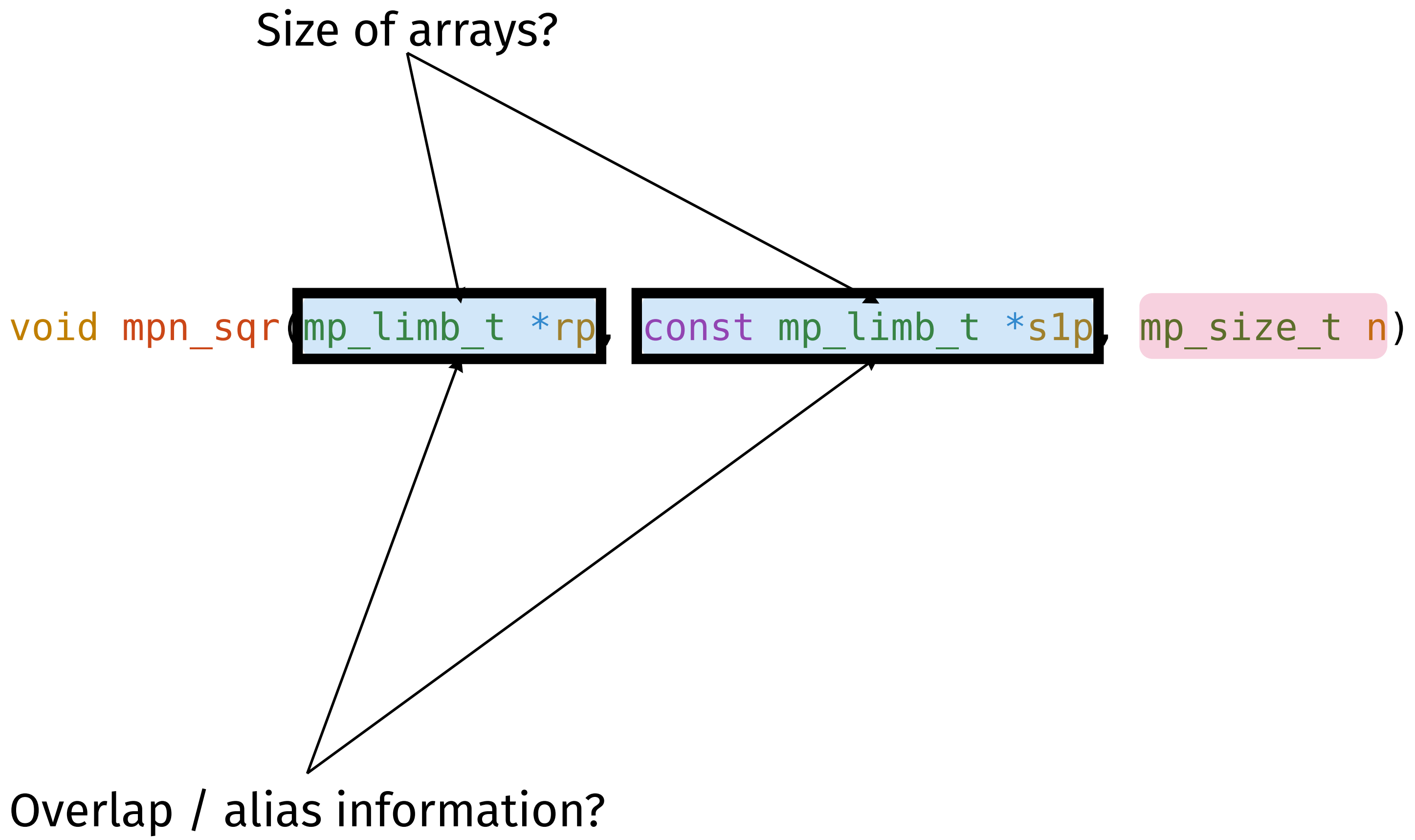
Lack of expressiveness in C signatures



Size of arrays?

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

Lack of expressiveness in C signatures



Size of arrays?

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

Overlap / alias information?

Lack of expressiveness in C signatures

Size and alias information may be specified in the documentation:

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

«

- Size of `rp` and `s1p` are respectively 2`n` and `n`.
- No overlap between `rp` and `s1p`.

»

Lack of expressiveness in C signatures

Size and alias information may be specified in the documentation:

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

«
- Size of `rp` and `s1p` are respectively 2`n` and `n`.
- No overlap between `rp` and `s1p`.

»

But the C compiler cannot enforce these requirements!

Expressive Capla signatures

In C:

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

In Capla:

```
fun mpn_sqr(rp: mut [u64; 2 * n], s1p: [u64; n], n: u64)
```

Expressive Capla signatures

In C:

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

In Capla:

```
fun mpn_sqr(rp: mut [u64; 2 * n], s1p: [u64; n], n: u64)
```

The semantics of Capla brings two guarantees:

- `rp` and `s1p` have the specified size.
- `rp` is `mut` ⇒ it cannot alias with `s1p`.

Expressive Capla signatures

In C:

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

In Capla:

```
fun mpn_sqr(rp: mut [u64; 2 * n], s1p: [u64; n], n: u64)
```

The semantics of Capla brings two guarantees:

- `rp` and `s1p` have the specified size.
- `rp` is `mut` ⇒ it cannot alias with `s1p`.

Remark: The signatures of C and Capla are compatible.

Lack of expressiveness in Fortran signatures

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

Same problem.

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

Lack of expressiveness in Fortran signatures

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

Same problem.

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

In Capla:

```
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])
```

Dynamic tests in Capla

```
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 {
    ...
  } }
```

Straightforward translation from the original BLAS implementation in Fortran

Dynamic tests in Capla

```
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 {
    ...
  } }
```

Dynamic test: $1 < 2$

Straightforward translation from the original BLAS implementation in Fortran

Dynamic tests in Capla

```
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 {
    ...
  } }
```

Dynamic test: $1 < 2$

Trivially eliminated

Straightforward translation from the original BLAS implementation in Fortran

Dynamic tests in Capla

```
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 {
    ...
  } }
```

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

Can be eliminated

Dynamic test: $1 < 2$

Trivially eliminated

Straightforward translation from the original BLAS implementation in Fortran

Copy-restore and aliasing

- **Pointer-free** semantics: no global memory, only local environments
  - Arrays are always copied and never shared when calling functions

Copy-restore and aliasing

- **Pointer-free** semantics: no global memory, only local environments
  - Arrays are always copied and never shared when calling functions
  - 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
}
```

Copy-restore and aliasing

- **Pointer-free** semantics: no global memory, only local environments
  - Arrays are always copied and never shared when calling functions
  - Non-mutable variables are trivially unmodified
  - 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
}
```

Copy-restore and aliasing

- **Pointer-free** semantics: no global memory, only local environments
  - Arrays are always copied and never shared when calling functions
  - Non-mutable variables are trivially unmodified
  - Program verification: no need for separation logic for proofs

- 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;
}
```

Copy-restore and aliasing

- **Pointer-free** semantics: no global memory, only local environments
  - Arrays are always copied and never shared when calling functions
  - Non-mutable variables are trivially unmodified
  - Program verification: no need for separation logic for proofs

- Efficient generated programs: use pointers instead of deep copies
  - **Non-aliasing policy**: mutable arrays are separated from any other array

```
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 polynomial multiplication)

```
fun karatsuba(r: mut [i64; 2 * n], a b: [i64; n],
              t: mut [i64; 2 * n], n: u64)
{ ...
  let 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);
  sub2(t, r, 2 * k);
  sub2(t, r[(2 * k)..], 2 * k); }
  add2(r[k..], t, 2 * k);
}
```

$$\begin{aligned}
r_0 &\leftarrow a_0 + a_1 \\
r_2 &\leftarrow b_0 + b_1 \\
t &\leftarrow r_0 r_2 = (a_0 + a_1)(b_0 + b_1) \\
r_{0,1} &\leftarrow a_0 b_0 \\
r_{2,3} &\leftarrow a_1 b_1 \\
t &\leftarrow t - r_{0,1} \\
t &\leftarrow t - r_{2,3} = a_1 b_0 + a_0 b_1 \\
r_{1,2} &\leftarrow r_{1,2} + t
\end{aligned}$$

Non-aliasing policy and pointer arithmetic (Karatsuba polynomial multiplication)

```
fun karatsuba(r: mut [i64; 2 * n], a b: [i64; n],
              t: mut [i64; 2 * n], n: u64)
{ ...
  let 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);
  sub2(t, r, 2 * k);
  sub2(t, r[(2 * k)..], 2 * k); }
  add2(r[k..], t, 2 * k);
}
```

Alias between mutable arrays

Non-aliasing policy and pointer arithmetic (Karatsuba polynomial multiplication)

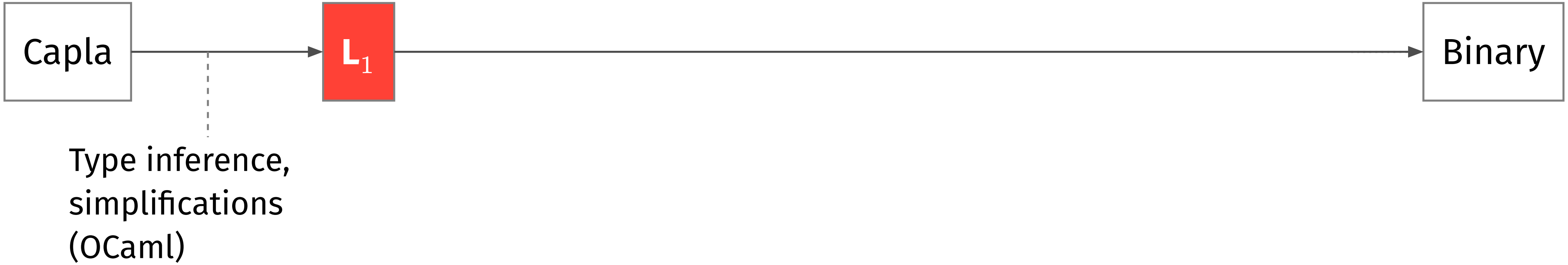
```
fun karatsuba(r: mut [i64; 2 * n], a b: [i64; n],
              t: mut [i64; 2 * n], n: u64)
{ ...
  let k = n / 2;
  let [a0: ..k; a1: k..] = a;
  let [b0: ..k; b1: k..] = b;
  let [t0: ..(2 * k); t1: ..] = t;
  { let [r0: ..(2 * k); r2: ..] = r;
    add(r0[..k], a0, a1, k);
    add(r2[..k], b0, b1, k);
    karatsuba(t0, r0[..k], r2[..k], t1, k);
    karatsuba(r0, a0, b0, t1, k);
    karatsuba(r2, a1, b1, t1, k);
    sub2(t0, r0, 2 * k);
    sub2(t0, r2, 2 * k); }
  add2(r[k..(3 * k)], t0, 2 * k);
}
```

← t is restored here

Copy-restore semantics

Capla
Binary

Copy-restore semantics

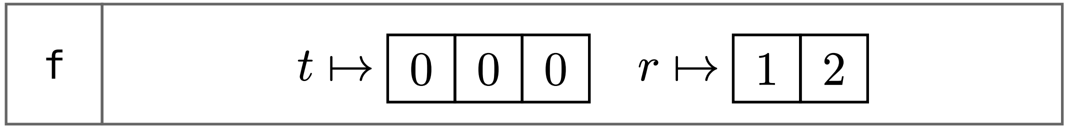


Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
→ g(t);
  ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
    let s = alloc i32, 2;
  }
}
```

Local environments



Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
  ...
}

fun g(u: mut [i64; 3]) {
→ { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
    let s = alloc i32, 2;
  }
}
```

Local environments

g	$u \mapsto \boxed{0}\boxed{0}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $r \mapsto \boxed{1}\boxed{2}$

Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
  ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
  → u2[0] = 1;
    let s = alloc i32, 2;
  }
}
```

Local environments

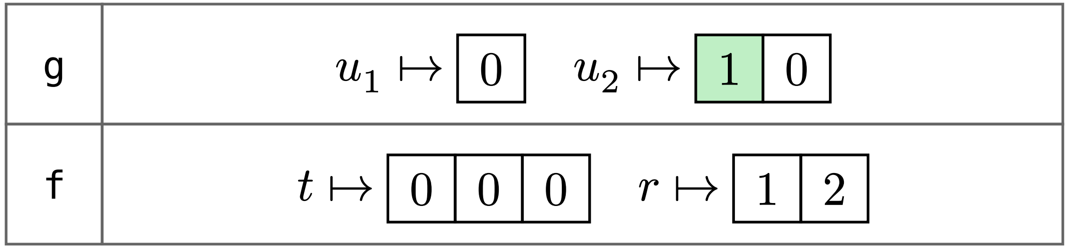
g	$u_1 \mapsto \begin{array}{\|c\|}\hline 0 \\ \hline\end{array}$ $\quad u_2 \mapsto \begin{array}{\|c\|c\|}\hline 0 & 0 \\ \hline\end{array}$
f	$t \mapsto \begin{array}{\|c\|c\|c\|}\hline 0 & 0 & 0 \\ \hline\end{array}$ $\quad r \mapsto \begin{array}{\|c\|c\|}\hline 1 & 2 \\ \hline\end{array}$

Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
  ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
  → let s = alloc i32, 2;
  }
}
```

Local environments



g	$u_1 \mapsto \boxed{0}$ $u_2 \mapsto \boxed{1}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $r \mapsto \boxed{1}\boxed{2}$

Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
  ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
    let s = alloc i32, 2;
→ }
}
```

Local environments

g	$u_1 \mapsto \boxed{0}$ $u_2 \mapsto \boxed{1}\boxed{0}$ $s \mapsto \boxed{0}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $r \mapsto \boxed{1}\boxed{2}$

Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
  ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
    let s = alloc i32, 2;
  }
→ }
```

Local environments

g	$u \mapsto$	0	1	0	$s \mapsto$	0	0
f	$t \mapsto$	0	0	0	$r \mapsto$	1	2

Copy-restore

Programs behave as if arrays were deep-copied at function entry and copied back at the end of the function.

```
fun f(t: mut [i64; 3], r: [i64; 2]) {
  g(t);
→ ...
}

fun g(u: mut [i64; 3]) {
  { let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
    let s = alloc i32, 2;
  }
}
```

Local environments

f	$t \mapsto$	0	1	0	$r \mapsto$	1	2

Writing into an array

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

Local environments: $E : x \mapsto v$

State: $(\text{env}, \text{sizes}, \langle\text{code}\rangle, \text{continuation})$

Size environments: $S : x \mapsto n$ 💡

Writing into an array

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

Local environments: $E : x \mapsto v$

State: $(\mathrm{env}, \mathrm{sizes}, \langle \mathrm{code} \rangle, \mathrm{continuation})$

Size environments: $S : x \mapsto n$

$$\frac{\begin{array}{c} E, S \vdash e \Rightarrow \texttt{Vint}_{64} n \\ E, S \vdash e' \Rightarrow v \end{array}}{(E, S, \langle\, x[e] = e' \,\rangle, k) \rightarrow (\, E[x \leftarrow E(x)[n \leftarrow v]] \,, S, \langle \texttt{skip} \rangle, k)}\ \textsc{Write}$$

🟨 Statically guaranteed by typing

🟥 Dynamically tested

Writing into an array

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

State: $(\text{env}, \text{sizes}, \langle\text{code}\rangle, \text{continuation})$

Local environments: $E : x \mapsto v$

Size environments: $S : x \mapsto n$

$$\dfrac{\texttt{perm}(x) \geq \texttt{Mutable} \qquad E, S \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad E, S \vdash e' \Rightarrow v \qquad \texttt{primitive}(v)}{(E, S, \langle x[e] = e' \rangle, k) \rightarrow (E[x \leftarrow E(x)[n \leftarrow v]], S, \langle \texttt{skip} \rangle, k)}\ \textsc{Write}$$

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

🟨 Statically guaranted by typing

🟥 Dynamically tested

Writing into an array

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

Local environments: $E : x \mapsto v$

State: $(\text{env}, \text{sizes}, \langle\text{code}\rangle, \text{continuation})$

Size environments: $S : x \mapsto n$

$$\dfrac{\begin{array}{c} \texttt{perm}(x) \geq \texttt{Mutable} \qquad E, S \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad n < S(x) \\ E, S \vdash e' \Rightarrow v \qquad \texttt{primitive}(v) \end{array}}{(E, S, \langle x[e] = e' \rangle, k) \rightarrow (E[x \leftarrow E(x)[n \leftarrow v]], S, \langle \texttt{skip} \rangle, k)}\ \textsc{Write}$$

$$\dfrac{\begin{array}{c} \texttt{perm}(x) \geq \texttt{Mutable} \qquad E \vdash e \Rightarrow \texttt{Vint}_{64} n \qquad n \geq S(x) \\ E \vdash e' \Rightarrow v \qquad \texttt{primitive}(v) \end{array}}{(E, S, \langle x[e] = e' \rangle, k) \rightarrow (E, S, \langle \texttt{error} \rangle, k)}\ \textsc{WriteErr}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

```
fun g(a: mut [i64; 1]) {
  f(a);
}

fun f(x: [i64; 1]) {
  ...
}
```

$$\frac{E(\vec{a}) = \vec{v} \qquad \texttt{f.params} = \vec{x}}{(E, S, \langle \texttt{f}(\vec{a}) \rangle, k) \rightarrow}\ \textsc{Call}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

```
fun g(a: [i64; 1]) {
  let v = a[0];
  f(a); // illegal
        // otherwise, a would change
  assert(a[0] == v);
}

fun f(x: mut [i64; 1]) {
  x[0] = 1;
}
```

$$\frac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \texttt{f.params} = \vec{x} \\ \forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i) \end{array}}{(E, S, \langle \texttt{f}(\vec{a}) \rangle, k) \to} \ \textsc{Call}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

```
fun g(a: mut [i64; 1]) {
  f(a, a); // illegal
}

fun f(x: mut [i64; 1], y: [i64; 1]) {
  let v = y[0];
  x[0] = 1;
  // otherwise, y would change
  assert(y[0] == v);
}
```

$$\frac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \texttt{f}.\texttt{params} = \vec{x} \\ \forall i, \texttt{f}.\texttt{perm}(x_i) \leq \texttt{perm}(a_i) \\ \forall i\ j, \texttt{f}.\texttt{perm}(x_i) \geq \texttt{Mutable} \land i \neq j \Rightarrow a_i \neq a_j \end{array}}{(E, S, \langle \texttt{f}(\vec{a}) \rangle, k) \to}\ \textsc{Call}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

```
fun g(a: mut [i64; n], n: u64) {
  f(a, n + 1); // illegal
}

fun f(x: mut [i64; n]; n: u64) {
  x[n - 1] = 1; // out-of-bound access
}
```

$$\dfrac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \texttt{f.params} = \vec{x} \\ \forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i) \\ \forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \land i \neq j \Rightarrow a_i \neq a_j \\ S(\vec{a}) = S_{\texttt{f}}(\vec{x}) \\ \\ \end{array}}{(E, S, \langle \texttt{f}(\vec{a}) \rangle, k) \to}\ \textsc{Call}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

```
fun g(a: mut [i64; 1]) {
  f(a);
}

fun f(x: [i64; 1]) {
  ...
}
```

$$\dfrac{\begin{array}{c} E(\vec{a}) = \vec{v} \qquad \texttt{f.params} = \vec{x} \\ \forall i, \texttt{f.perm}(x_i) \leq \texttt{perm}(a_i) \\ \forall i\ j, \texttt{f.perm}(x_i) \geq \texttt{Mutable} \land i \neq j \Rightarrow a_i \neq a_j \\ S(\vec{a}) = S_{\texttt{f}}(\vec{x}) \\ E_{\texttt{f}} = \ldots \qquad S_{\texttt{f}} = \ldots \end{array}}{(E, S, \langle \texttt{f}(\vec{a}) \rangle, k) \to (E_{\texttt{f}}, S_{\texttt{f}}, \texttt{f.body}, \texttt{Kcall}(E, S, m, k))}\ \text{Call}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

Copy

$$\frac{\begin{array}{c}\mathtt{f.params} = \vec{x} \\ \dots \\ E_{\mathsf{f}} = \dots \\ m = \{(a_i, x_i) \mid \mathtt{f.perm}(x_i) = \mathtt{Mutable}\}\end{array}}{(E, S, \langle \mathtt{f}(\vec{a})\rangle, k) \to (E_{\mathsf{f}}, S_{\mathsf{f}}, \mathtt{f.body}, \mathtt{Kcall}(E, S, m, k))}\ \textsc{Call}$$

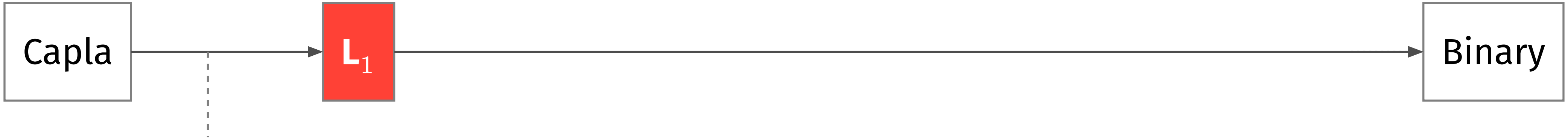
$$\frac{}{\begin{array}{c}(E_{\mathsf{f}}, S_{\mathsf{f}}, \langle \mathtt{return}\rangle, \mathtt{Kcall}(E, S, m, k)) \to \\ (E[a_i \leftarrow E_{\mathsf{f}}(x_i) \mid (a_i, x_i) \in m], S, \langle \mathtt{skip}\rangle, k)\end{array}}\ \textsc{Return}$$

Restore

🟨 Statically guaranted by typing

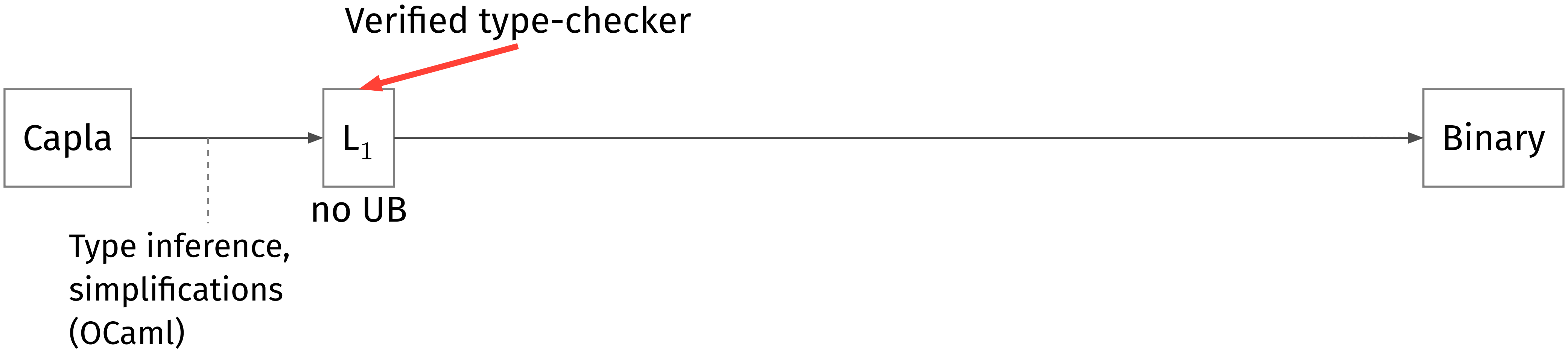
🟥 Dynamically tested

Typing and safety



Type inference,
simplifications
(OCaml)

Typing and safety



Preventing UB by typing

- Accesses to uninitialized variables

```
let x: i32;
if (b) { x = 0; }
y = x;
```

UB and rejected

```
let x: i32;
if (b) { x = 0; }
if (b) { y = x; }
```

Safe but rejected

Preventing UB by typing

- Accesses to uninitialized variables

```
let x: i32;
if (b) { x = 0; }
y = x;
```

UB and rejected

```
let x: i32;
if (b) { x = 0; }
if (b) { y = x; }
```

Safe but rejected

- Aliases in function calls (*e.g.*, arrays of arrays)

```
fun f(u: mut [i64; 42], v: [i64; 42]) { ... }
fun g(t: mut [[i64; 42]; n], n: u64) {
```

```
let j = i;
f(t[i], t[j]);
```

UB and rejected

```
let j = i + 1;
f(t[i], t[j]);
```

Safe but rejected

```
f(t[0], t[1]);
```

Safe and accepted

Type safety

Theorem (Type safety)

ROCQ APPROVED

~6000 loc

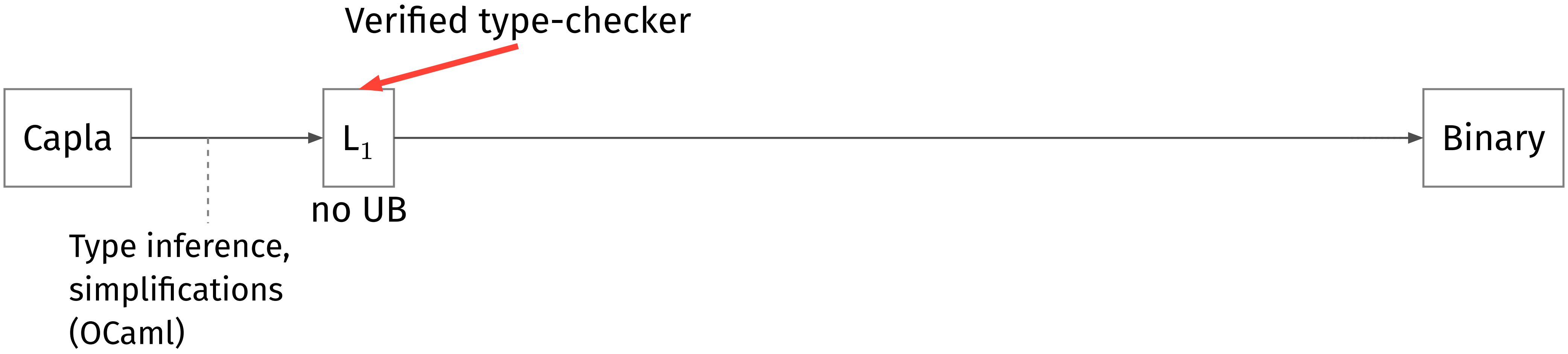
Given a successfully typed L_1 program,
if all the invariants hold in a non-final L_1 state s, then

- *there exists a state t such that $s \to t$ and,*
- *for every t such that $s \to t$, all the invariants hold in t.*

Invariants

1. The content of the local environment has the expected type.
2. The needed variables are initialized.
3. Size environments are well-formed.
4. Multidimensional arrays have valid sizes: $\texttt{length}(E(t)) = S(t)$.

Compiler and semantics preservation



Verified type-checker

Capla

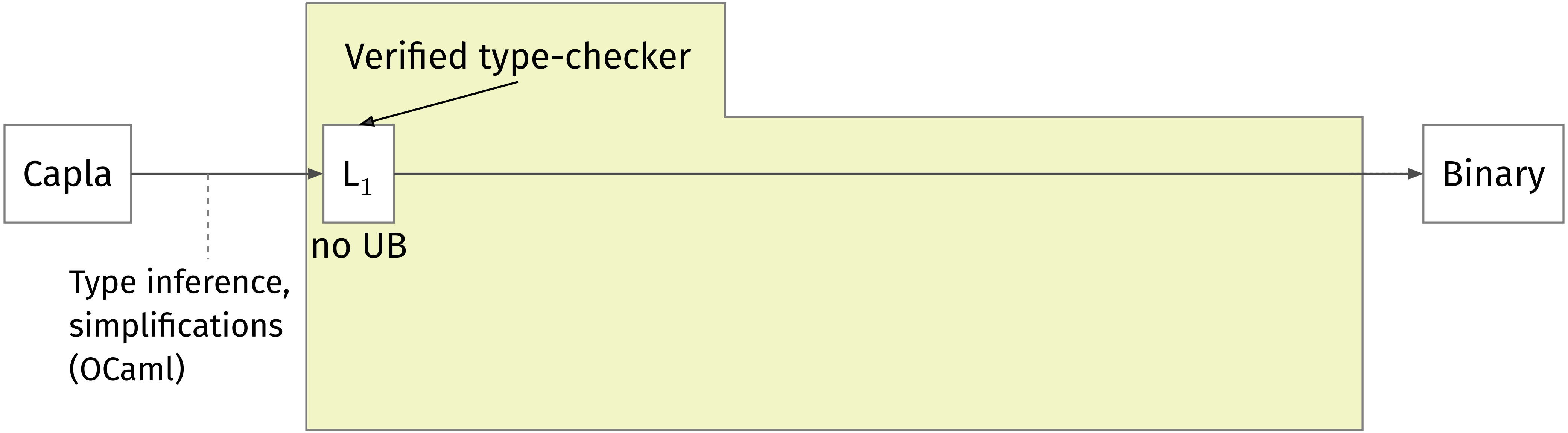
L_1

Binary

no UB

Type inference,
simplifications
(OCaml)

Compiler and semantics preservation


Verified type-checker

Capla

L_1

no UB

Binary

Type inference,
simplifications
(OCaml)

Formally verified translation (Rocq)

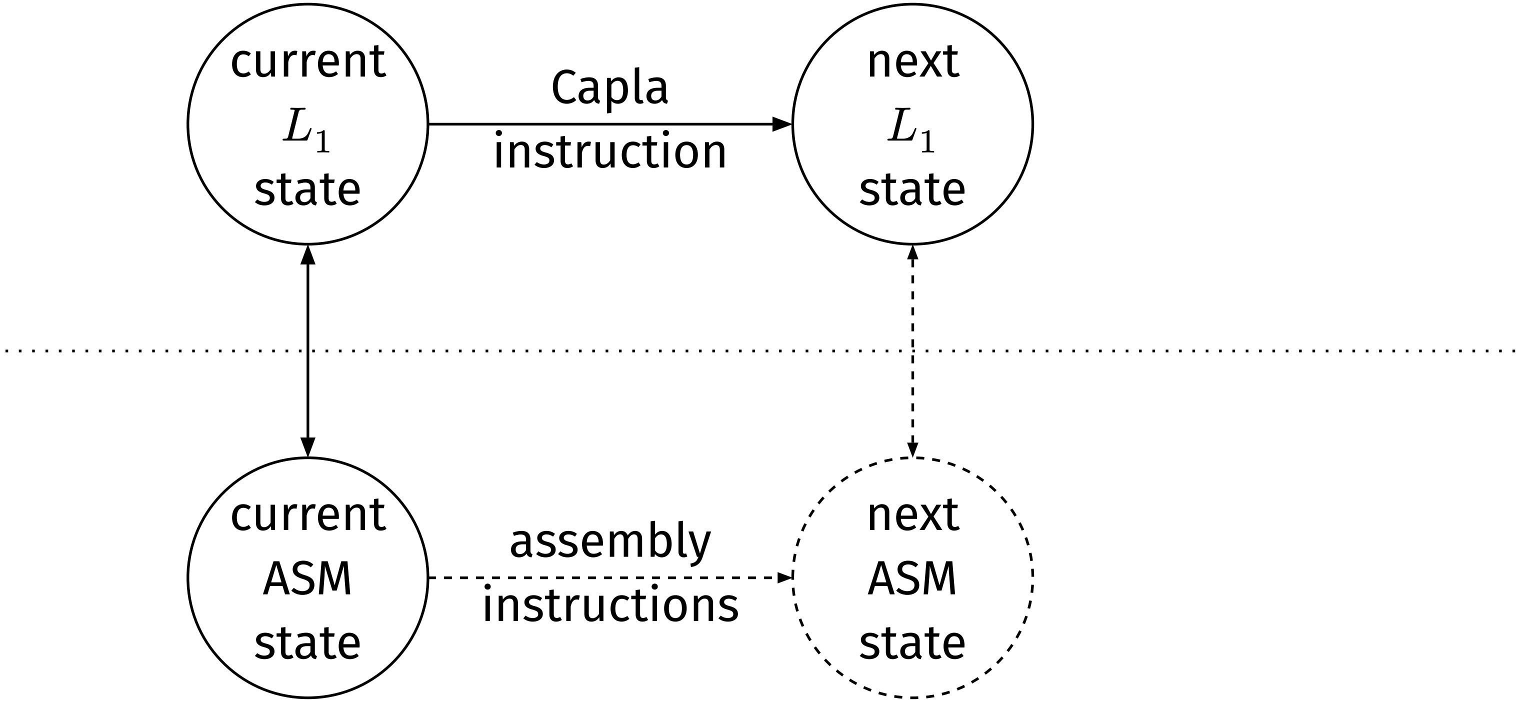
Capla compiler correctness

Theorem (Compiler correctness)

Let p be an L_1 program.
Assuming p has been successfully compiled to an ASM program p',
any behavior of p' is also a behavior of p, according to the semantics of L_1.

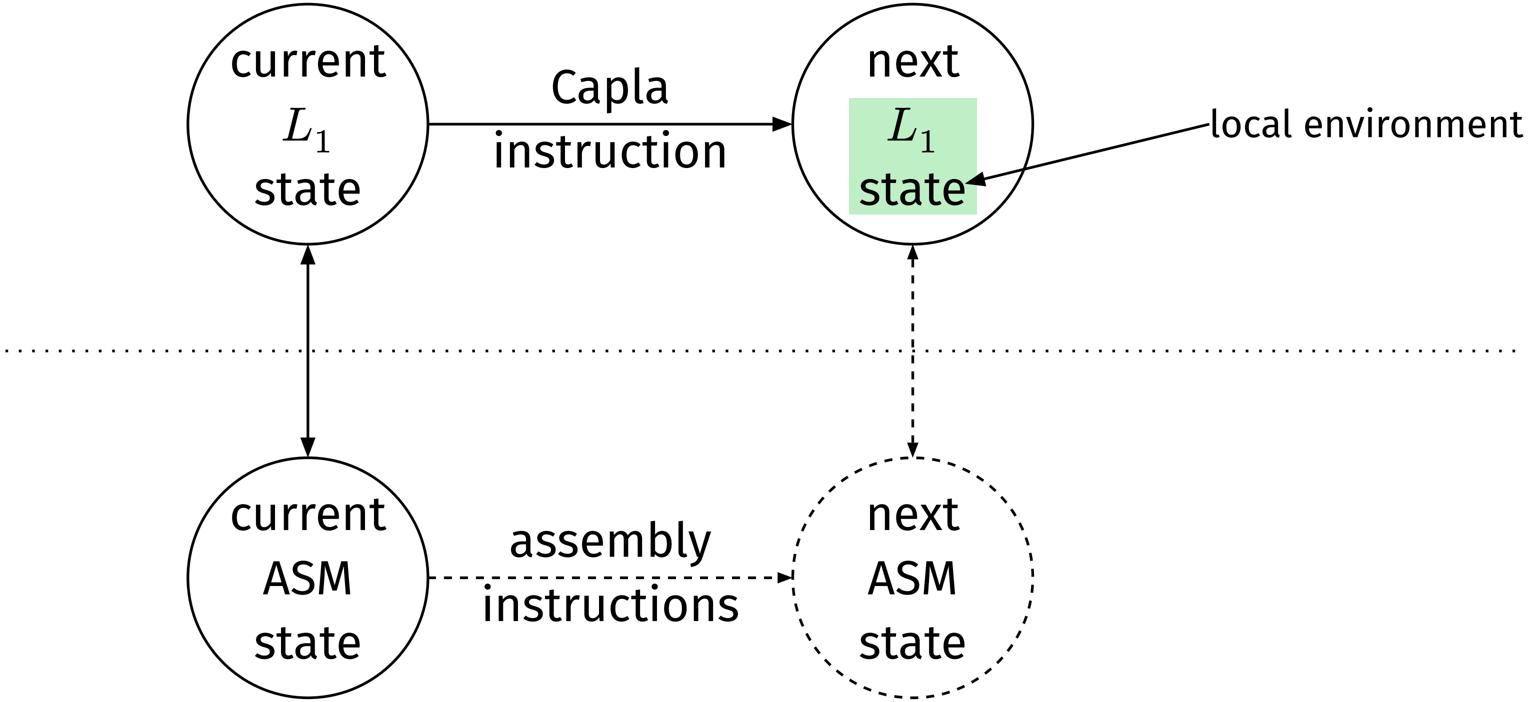
Forward simulation

Each step in Capla implies a sequence of no-UB steps in assembly and execution states are still related.



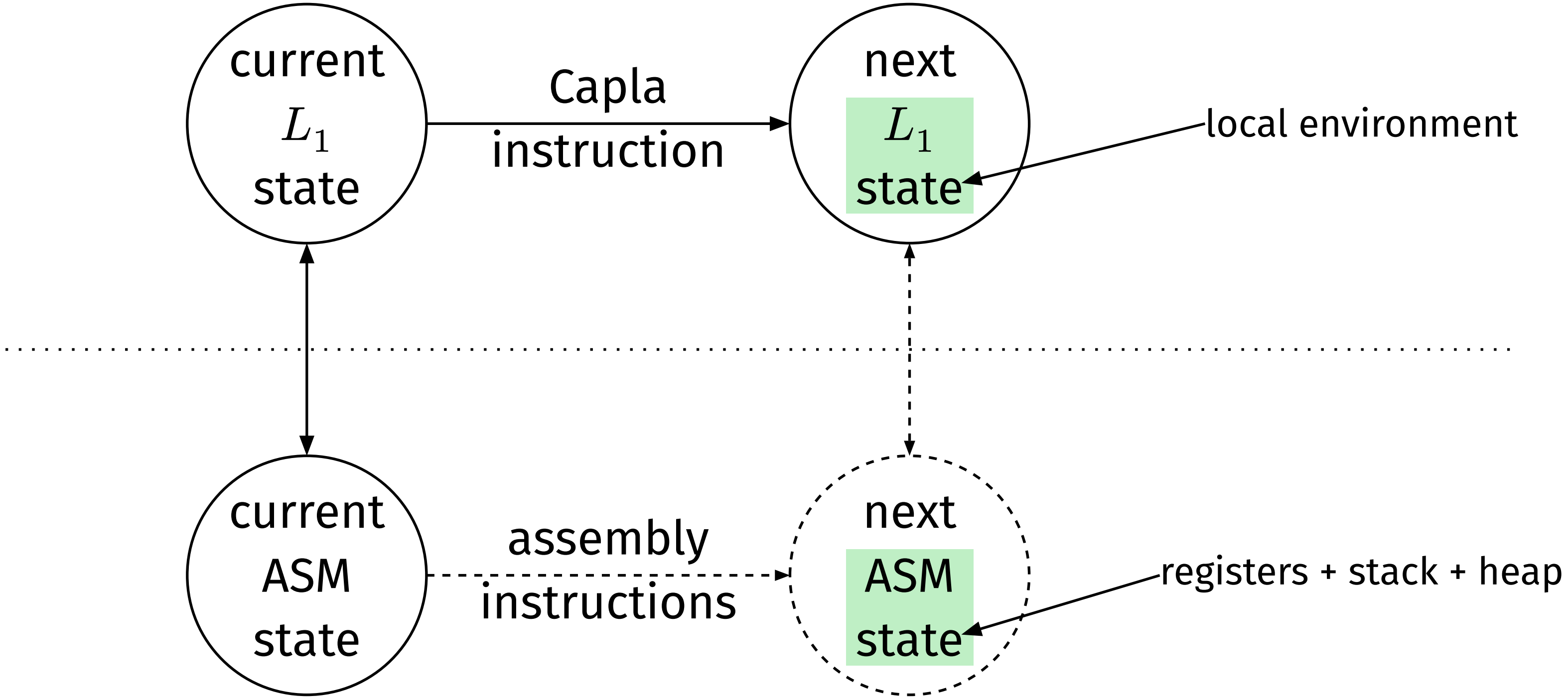
Forward simulation

Each step in Capla implies a sequence of no-UB steps in assembly and execution states are still related.

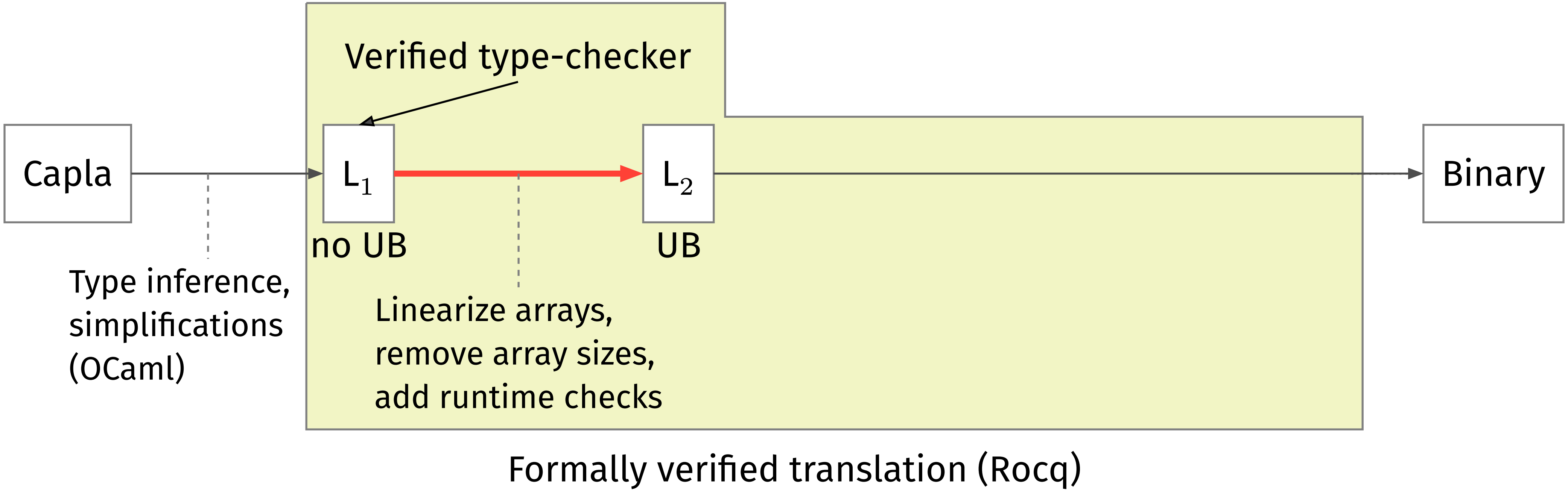


Forward simulation

Each step in Capla implies a sequence of no-UB steps in assembly and execution states are still related.



Compiler architecture



Translation from L_1 to L_2

```
fun g(u: mut [i64; 3]) {





  let [u1: ..; u2: 1..] = u;

  u2[0] = 1;

  let s = alloc i32, 2;
}
```

→

```
fun g(u: mut [i64]) -> void {
  u_size = 3u64;
  assert (1u64 <= u_size);
  u1_size = 1u64;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1u64;
  u1, u2 = split u __tmp {
    assert (0u64 < u2_size);
    u2[0u64] = 1i64;
    s_size = 2u64;
    alloc(s, s_size);
  } }
```

Translation from L_1 to L_2

```
fun g(u: mut [i64; 3]) {





  let [u1: ..; u2: 1..] = u;

  u2[0] = 1;

  let s = alloc i32, 2;
}
```

→

```
fun g(u: mut [i64]) -> void {
  u_size = 3u64;
  assert (1u64 <= u_size);
  u1_size = 1u64;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1u64;
  u1, u2 = split u __tmp {
    assert (0u64 < u2_size);
    u2[0u64] = 1i64;
    s_size = 2u64;
    alloc(s, s_size);
  } }
```

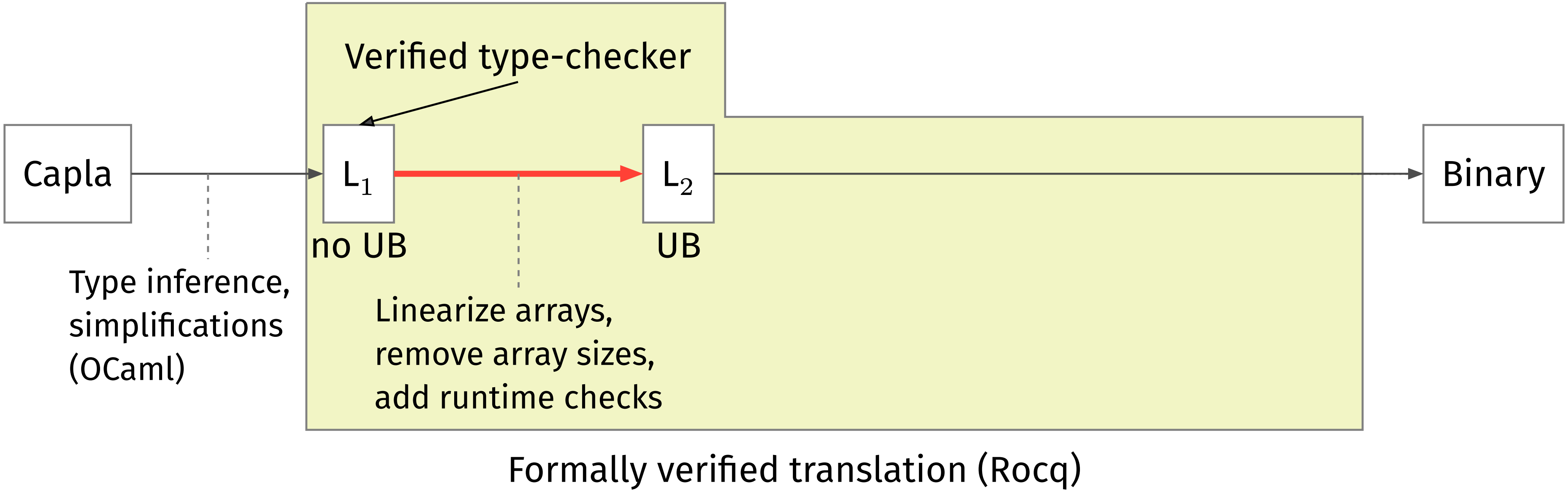
Semantics preservation from L_1 to L_2

	L_1 mathematical formulas	L_2 executable code
Tests must be correct and complete (*e.g.*, cast `f64` $\rightarrow$ `i32`)	$-2^{31} \leq \lfloor f \rceil < 2^{31}$	$-2^{31} - 1 < f < 2^{31}$

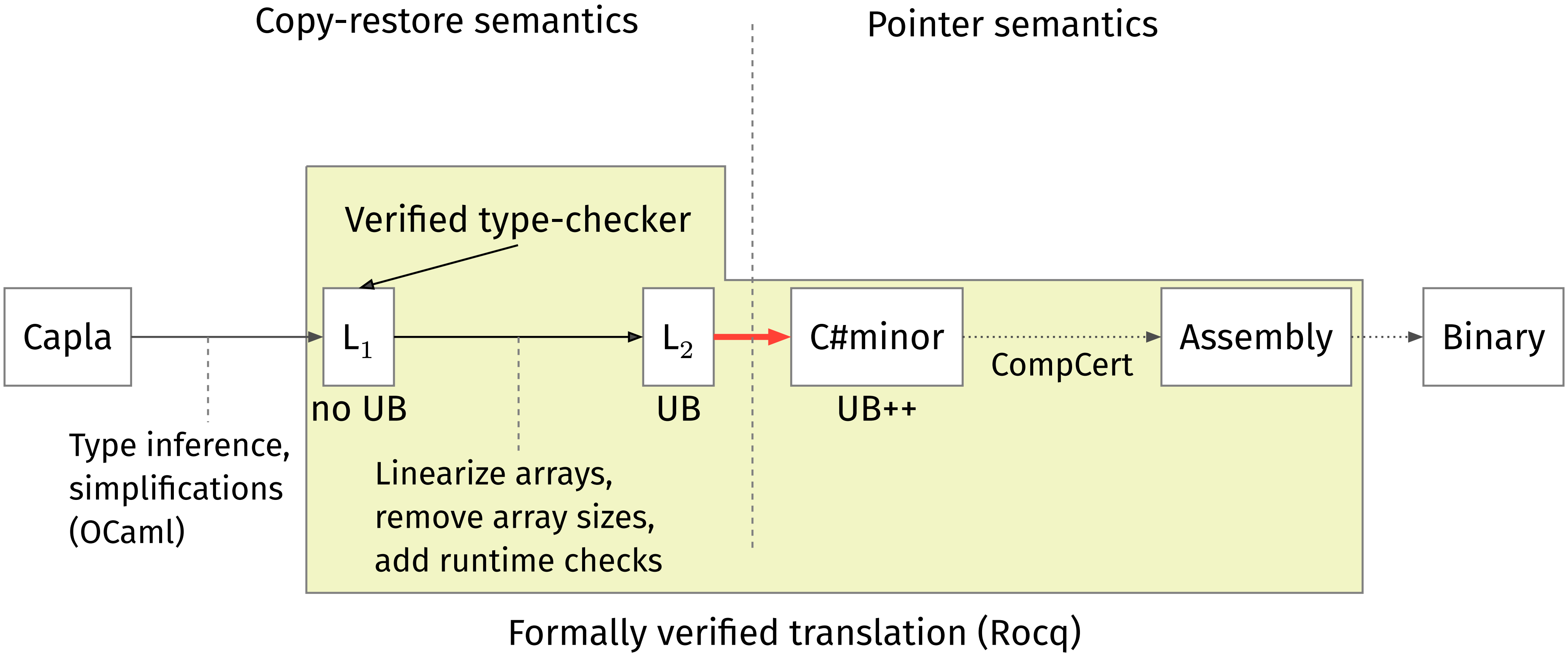
Semantics preservation from L_1 to L_2

	L_1 mathematical formulas	L_2 executable code
Tests must be correct and complete (*e.g.*, cast `f64` → `i32`)	$-2^{31} \leq \lfloor f \rceil < 2^{31}$	$-2^{31} - 1 < f < 2^{31}$
Array indices must be computed without overflows	$E, S \vdash t[x, y] \Rightarrow E(t)[i]$ with $i = E(x) \times_{\mathbb{Z}} S(t) +_{\mathbb{Z}} E(y)$	$t[x \times_{\mathsf{u64}} s +_{\mathsf{u64}} y]$

Compiler architecture



Compiler architecture



Copy-restore semantics

Pointer semantics

Verified type-checker

Capla

L_1

L_2

C#minor

Assembly

Binary

CompCert

no UB

UB

UB++

Type inference, simplifications (OCaml)

Linearize arrays, remove array sizes, add runtime checks

Formally verified translation (Rocq)

Translation from L_2 to C#minor

L_1

```
fun g(u: mut [i64; 3]) {






  let [u1: ..; u2: 1..] = u;

  u2[0] = 1;

  let s = alloc i32, 2;

}
```

→

L_2

```
fun g(u: mut [i64]) -> void {
  u_size = 3u64;
  assert (1u64 <= u_size);
  u1_size = 1u64;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1u64;

  u1, u2 = split u __tmp {
    assert (0u64 < u2_size);
    u2[0u64] = 1i64;
    s_size = 2u64;
    alloc(s, s_size);

  } }
```

→

C#minor

```
void g(int64_t* u) {
  u_size = 3LL;
  assert(1LL <= u_size);
  u1_size = 1LL;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1LL;
  u1 = u;
  u2 = u + 8LL * __tmp;
  assert(0LL < u2_size);
  *(u2 + 8LL * 0LL) = 1LL;
  s_size = 2LL;
  s = calloc(s_size, 4LL);
  assert(s != 0LL);
}
```

Translation from L_2 to C#minor

L_1

```
fun g(u: mut [i64; 3]) {






  let [u1: ..; u2: 1..] = u;

  u2[0] = 1;

  let s = alloc i32, 2;

}
```

→

L_2

```
fun g(u: mut [i64]) -> void {
  u_size = 2u64;
  assert (1u64 <= u_size);
  u1_size = 1u64;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1u64;

  u1, u2 = split u __tmp {
    assert (0u64 < u2_size);
    u2[0u64] = 1i64;
    s_size = 2u64;
    alloc(s, s_size);

  } }
```

→

C#minor

```
void g(int64_t* u) {
  u_size = 3LL;
  assert(1LL <= u_size);
  u1_size = 1LL;
  u2_size = u_size - u1_size;
  __tmp = u1_size * 1LL;
  u1 = u;
  u2 = u + 8LL * __tmp;
  assert(0LL < u2_size);
  *(u2 + 8LL * 0LL) = 1LL;
  s_size = 2LL;
  s = calloc(s_size, 4LL);
  assert(s != 0LL);
}
```

Relation between the memory models of L_2 and C#minor

L_2

Local environments

f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $\quad r \mapsto \boxed{1}\boxed{2}$

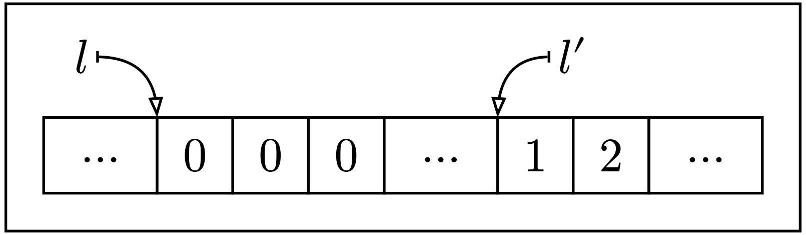
```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
→   g(t);
  }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
  }
```

C#minor

Local environments

f	$t \mapsto$ `Vptr` l $\quad r \mapsto$ `Vptr` l'

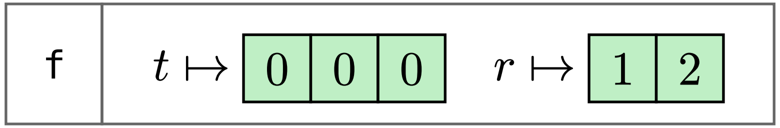
Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

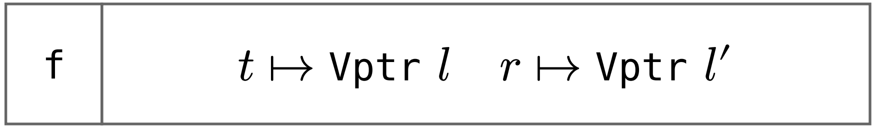


f | $t \mapsto$ 0 0 0 | $r \mapsto$ 1 2

```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
→   g(t);
  }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
  }
```

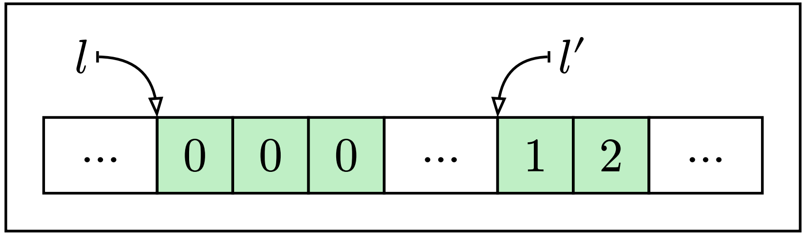
C#minor

Local environments



f | $t \mapsto$ `Vptr` l $r \mapsto$ `Vptr` l'

Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

g	$u \mapsto$ 🟩0 🟩0 🟩0
f	$t \mapsto$ 0 | 0 | 0 $r \mapsto$ 1 | 2

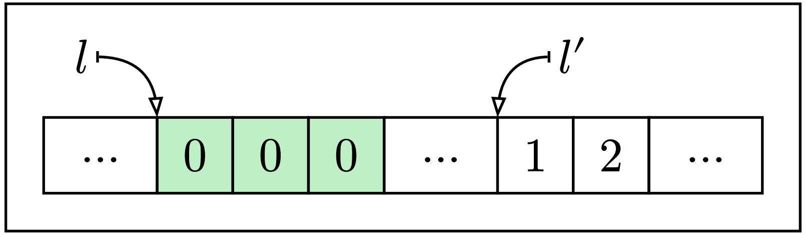
```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
    g(t);
  }

  fun g(u: mut [i64; 3]) {
→   let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
  }
```

C#minor

Local environments

g	$u \mapsto$ `Vptr` l
f	$t \mapsto$ `Vptr` l $r \mapsto$ `Vptr` l'

Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

g	$u_1 \mapsto \boxed{0} \quad u_2 \mapsto \boxed{0}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0} \quad r \mapsto \boxed{1}\boxed{2}$

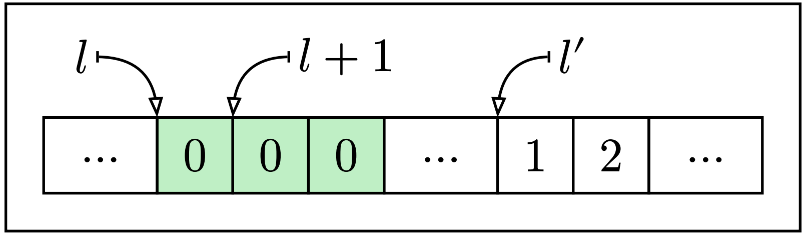
```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
    g(t);
  }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
→   u2[0] = 1;
  }
```

C#minor

Local environments

g	$u_1 \mapsto$ `Vptr` $l \quad u_2 \mapsto$ `Vptr` $(l+1)$
f	$t \mapsto$ `Vptr` $l \quad r \mapsto$ `Vptr` l'

Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

g	$u_1 \mapsto \boxed{0}$ $u_2 \mapsto \begin{array}{\|c\|c\|}\hline 1 & 0 \\ \hline\end{array}$
f	$t \mapsto \begin{array}{\|c\|c\|c\|}\hline 0 & 0 & 0 \\ \hline\end{array}$ $r \mapsto \begin{array}{\|c\|c\|}\hline 1 & 2 \\ \hline\end{array}$

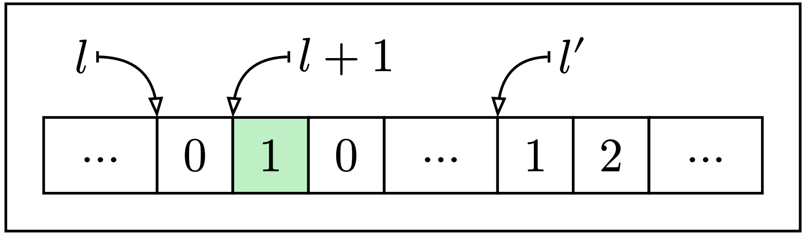
```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
    g(t);
  }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
→ }
```

C#minor

Local environments

g	$u_1 \mapsto$ `Vptr` l $u_2 \mapsto$ `Vptr` $(l+1)$
f	$t \mapsto$ `Vptr` l $r \mapsto$ `Vptr` l'

Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

g	$u \mapsto \boxed{0}\boxed{1}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $r \mapsto \boxed{1}\boxed{2}$

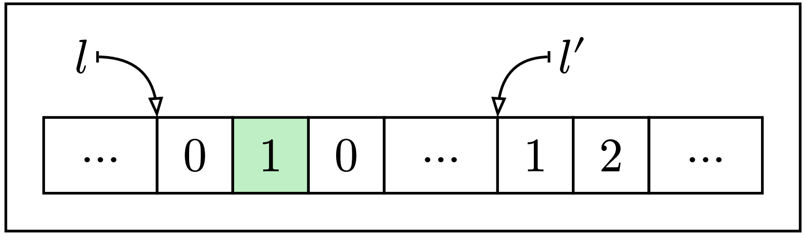
```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
    g(t);
  }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
→ }
```

C#minor

Local environments

g	$u \mapsto$ `Vptr` l
f	$t \mapsto$ `Vptr` l $r \mapsto$ `Vptr` l'

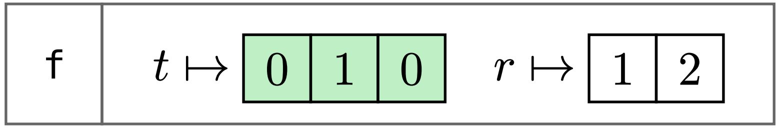
Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments



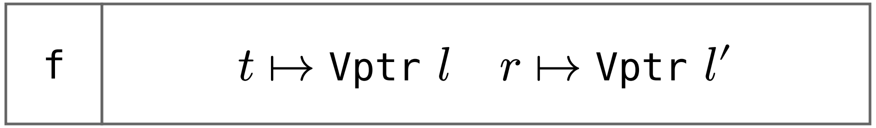
f

$t \mapsto$ 0 1 0 $r \mapsto$ 1 2

```
  fun f(t: mut [i64; 3], r: [i64; 2]) {
    g(t);
→ }

  fun g(u: mut [i64; 3]) {
    let [u1: ..; u2: 1..] = u;
    u2[0] = 1;
  }
```

C#minor

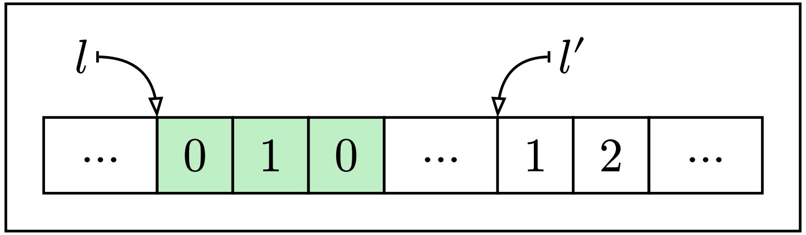
Local environments



f

$t \mapsto$ `Vptr` l $r \mapsto$ `Vptr` l'

Memory



Relation between the memory models of L_2 and C#minor

L_2

Local environments

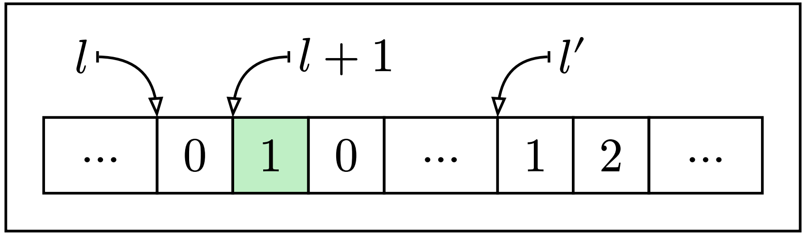
g	$u_1 \mapsto \boxed{0}$ $\quad u_2 \mapsto \boxed{1}\boxed{0}$
f	$t \mapsto \boxed{0}\boxed{0}\boxed{0}$ $\quad r \mapsto \boxed{1}\boxed{2}$

C#minor

Local environments

g	$u_1 \mapsto \texttt{Vptr}\ l \quad u_2 \mapsto \texttt{Vptr}\ (l+1)$
f	$t \mapsto \texttt{Vptr}\ l \quad r \mapsto \texttt{Vptr}\ l'$

Memory



Problem:

- The L_2 environment of the calling function does not match C#minor's memory.

Solution:

- Virtually performing return resynchronizes L_2 environments and C#minor's memory.

Capla compiler correctness

Theorem (Forward simulation)

ROCQ APPROVED

$L_1 \to L_2$: ~5000 loc
$L_2 \to C\#\mathrm{minor}$: ~9000 loc

Let $s \underset{L_1}{\to} t$, then for every state s' s.t. $s \Leftrightarrow s'$, either

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

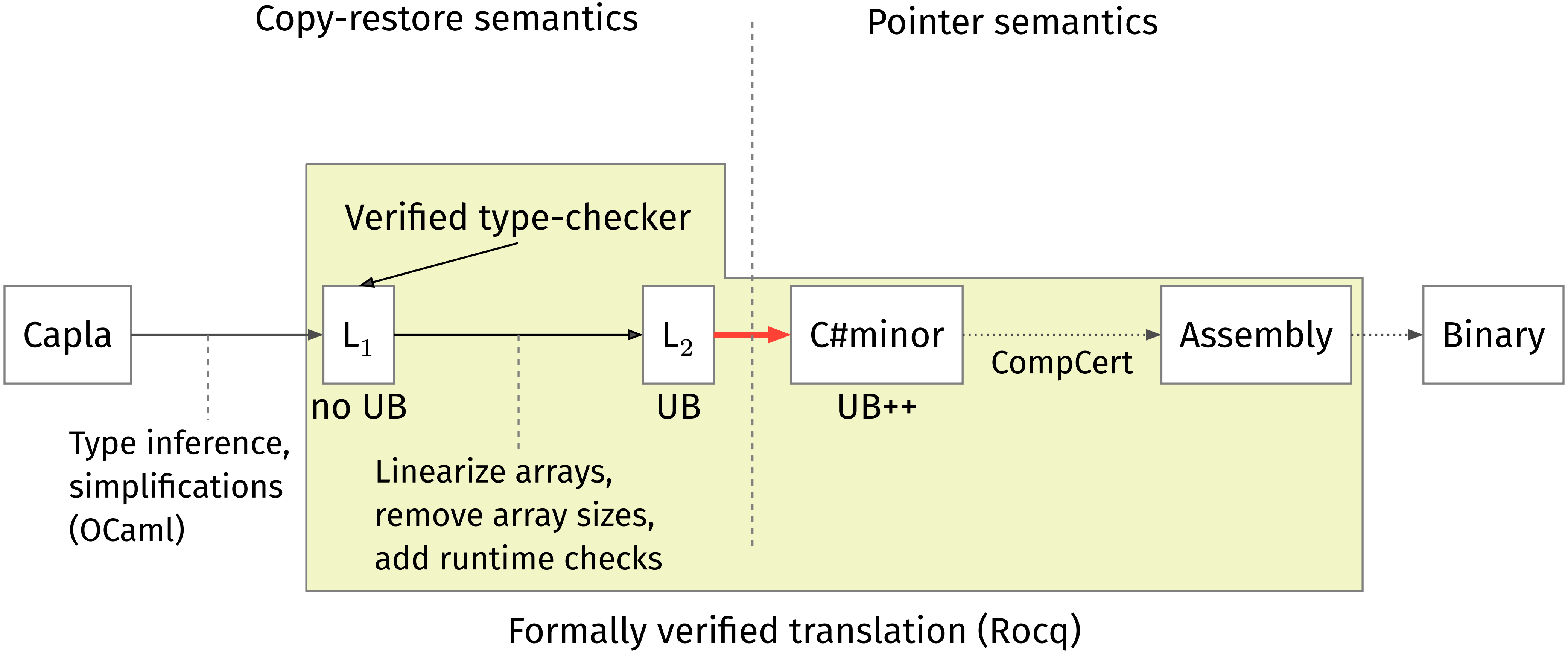
Corollary (Backward simulation)

ROCQ APPROVED

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

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

Expressiveness and benchmarks



Reimplementing some BLAS and GMP functions in Capla

BLAS

`saxpy`	$x \leftarrow \alpha x + y$	`zdotu`	$x \cdot y$
`sgemv dgemv`	$x \leftarrow \alpha Ax + \beta y$	`dtrsv`	$x \leftarrow A^{-1}x$ or $x \leftarrow A^{-T}x$

GMP

`mpn_cmp`	$x = y?$	`mpn_zero`	$x \leftarrow 0$
`mpn_add_1`	$y \leftarrow x + \alpha$	`mpn_add`	$z \leftarrow x + y$ with $\lvert x\rvert \geq \lvert y\rvert$
`mpn_add_n`	$z \leftarrow x + y$ with $\lvert x\rvert = \lvert y\rvert$	`mpn_add_nc`	$z \leftarrow x + y + \alpha$
`mpn_sub_1`	$y \leftarrow x - \alpha$	`mpn_sub`	$z \leftarrow x - y$ with $\lvert x\rvert \geq \lvert y\rvert$
`mpn_sub_n`	$z \leftarrow x - y$ with $\lvert x\rvert = \lvert y\rvert$	`mpn_sub_nc`	$z \leftarrow x - y - \alpha$
`mpn_lshift`	$y \leftarrow x \ll \alpha$	`mpn_rshift`	$y \leftarrow x \gg \alpha$
`mpn_addmul_1`	$y \leftarrow y + \alpha x$	`mpn_submul_1`	$y \leftarrow y - \alpha x$
`mpn_mul_1`	$y \leftarrow \alpha x$	`mpn_mul_basecase`	$z \leftarrow xy$ (schoolbook mul)
`mpn_toom22_mul`	$z \leftarrow xy$ (Toom-2 mul)	`mpn_toom32_mul`	$z \leftarrow xy$ (Toom-2.5 mul)
`mpn_mul_n`	$z \leftarrow xy$ when $\lvert x\rvert = \lvert y\rvert$	`mpn_mul`	$z \leftarrow xy$

Reimplementing some GMP functions

GMP generic C code

```
mp_limb_t mpn_addmul_1 (mp_ptr rp,
                        mp_srcptr up,
                mp_size_t n, mp_limb_t v0)
{
  mp_limb_t u0, crec, c, p1, p0, r0;
  ASSERT (n >= 1);
  ASSERT (MPN_SAME_OR_SEPARATE_P (rp, up, n));
  crec = 0;
  do {
    u0 = *up++;
    umul_ppmm (p1, p0, u0, v0);


    r0 = *rp;

    p0 = r0 + p0;
    c = r0 > p0;
    p1 = p1 + c;
    r0 = p0 + crec;
    c = p0 > r0;

    crec = p1 + c;
    *rp++ = r0;
  } while (--n != 0);

  return crec;
}
```

Capla

```
fun mpn_addmul_1_b(rp: mut [u64; n],
                   up:     [u64; n],
                   n v0: u64) -> u64
{
  let u0 crec c p1 p0 r0: u64;
  assert (n >= 1);

  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;
}
```

Reimplementing some GMP functions

GMP generic C code

```
mp_limb_t mpn_addmul_1 (mp_ptr rp,
                        mp_srcptr up,
                mp_size_t n, mp_limb_t v0)
{
  mp_limb_t u0, crec, c, p1, p0, r0;
  ASSERT (n >= 1);
  ASSERT (MPN_SAME_OR_SEPARATE_P (rp, up, n));
  crec = 0;
  do {
    u0 = *up++;
    umul_ppmm (p1, p0, u0, v0);


    r0 = *rp;

    p0 = r0 + p0;
    c = r0 > p0;
    p1 = p1 + c;
    r0 = p0 + crec;
    c = p0 > r0;

    crec = p1 + c;
    *rp++ = r0;
  } while (--n != 0);

  return crec;
}
```

Capla

```
fun mpn_addmul_1_b(rp: mut [u64; n],
                   up:     [u64; n],
                   n v0: u64) -> u64
{
  let u0 crec c p1 p0 r0: u64;
  assert (n >= 1);

  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;
}
```

GMP handwritten ASM (actually executed)

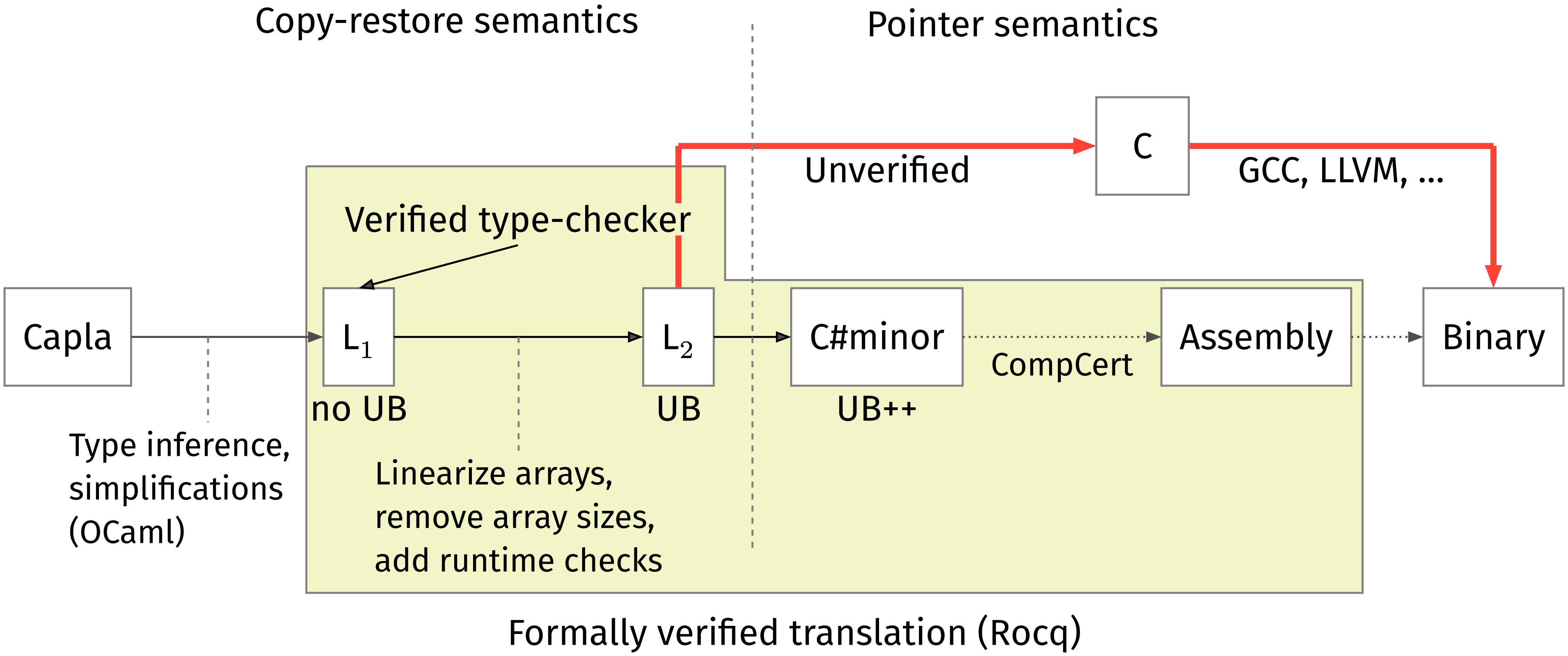
```
MULFUNC_PROLOGUE(mpn_addmul_1)
...
L(top): mul     %rcx
        add     %r8, %r10
        lea     (%rax), %r8
        mov     (up,%rbx,8), %rax
        adc     %r9, %r11
        mov     %r10, -8(rp,%rbx,8)
        mov     (rp,%rbx,8), %r10
        lea     (%rdx), %r9
        adc     $0, %rbp
L(mid): mul     %rcx
        add     %r11, %r10
        lea     (%rax), %r11
        mov     8(up,%rbx,8), %rax
        adc     %rbp, %r8
        mov     %r10, (rp,%rbx,8)
        mov     8(rp,%rbx,8), %r10
        lea     (%rdx), %rbp
        adc     $0, %r9
L(e):   add     $2, %rbx
        js      L(top)
...
```

Benchmarks

		Reference implementation	Capla reimplementation
BLAS	`saxpy`	1	4.58
	`zdotu`	1	2.26
	`sgemv`	1	1.71
	`dgemv`	1	2.86
	`dtrsv` (N)	1	2.07
	`dtrsv` (T)	1	2.10
GMP	`mpn_add_n`	1	4.77
	`mpn_addmul_1`	1	2.83
	`mpn_mul` (+ its deps)	1	3.26
	`mpn_mul` (only)	1	1.19

- Reference impl: x86-64 LAPACK 3.12.0 (Fortran) and GMP 6.3.0 (handwritten assembly)

Unverified C output



Benchmarks

		Capla reimplementation		
		CompCert	GCC	LLVM
BLAS	`saxpy`	4.58	3.52	0.76
	`zdotu`	2.26	2.25	1.16
	`sgemv`	1.71	1.14	0.31
	`dgemv`	2.86	1.26	0.89
	`dtrsv` (N)	2.07	1.12	1.37
	`dtrsv` (T)	2.10	1.49	0.89
GMP	`mpn_add_n`	4.77	2.33	2.70
	`mpn_addmul_1`	2.83	1.20	1.07
	`mpn_mul` (+ its deps)	3.26	1.58	1.57
	`mpn_mul` (only)	1.19	1.02	1.03

- ratio ≥ 1: slower than the reference implementation
- Reference impl: x86-64 LAPACK 3.12.0 (Fortran) and GMP 6.3.0 (handwritten assembly)
- LLVM 19.1.7 and GCC 14.3.0, optimization level `-O2 -ftree-vectorize`

Conclusion

Part 1: the Capla language

- Suitable for low-level numerical libraries
- Safe, runtime errors
- Copy-restore semantics for easier program reasoning

Part 1: the Capla language

- Suitable for low-level numerical libraries
- Safe, runtime errors
- Copy-restore semantics for easier program reasoning

Future work:

- Allow more complicated size expressions

```
fun zdotu(zx: [f64; 1 + (n - 1) * abs(incx), ...)
```

Part 1: the Capla language

- Suitable for low-level numerical libraries
- Safe, runtime errors
- Copy-restore semantics for easier program reasoning

Future work:

- Allow more complicated size expressions

```
fun zdotu(zx: [f64; 1 + (n - 1) * abs(incx), ...)
```

- Add constructions to the language (*e.g.*, records)

Part 1: the Capla language

- Suitable for low-level numerical libraries
- Safe, runtime errors
- Copy-restore semantics for easier program reasoning

Future work:

- Allow more complicated size expressions

```
fun zdotu(zx: [f64; 1 + (n - 1) * abs(incx), ...)
```

- Add constructions to the language (*e.g.*, records)
- Allow more complex views (*e.g.*, even/odd indices)

Part 1: the Capla language

- Suitable for low-level numerical libraries
- Safe, runtime errors
- Copy-restore semantics for easier program reasoning

Future work:

- Allow more complicated size expressions

```
fun zdotu(zx: [f64; 1 + (n - 1) * abs(incx), ...)
```

- Add constructions to the language (*e.g.*, records)
- Allow more complex views (*e.g.*, even/odd indices)
- Add a bit of aliasing (*e.g.*, allowed alias in `mpn_add`)

Part 2: the Capla compiler

- Translates Capla code to assembly
  - Using CompCert as a backend
  - Efficient implementation of the copy-restore semantics
  - Interoperability between both Capla and C code

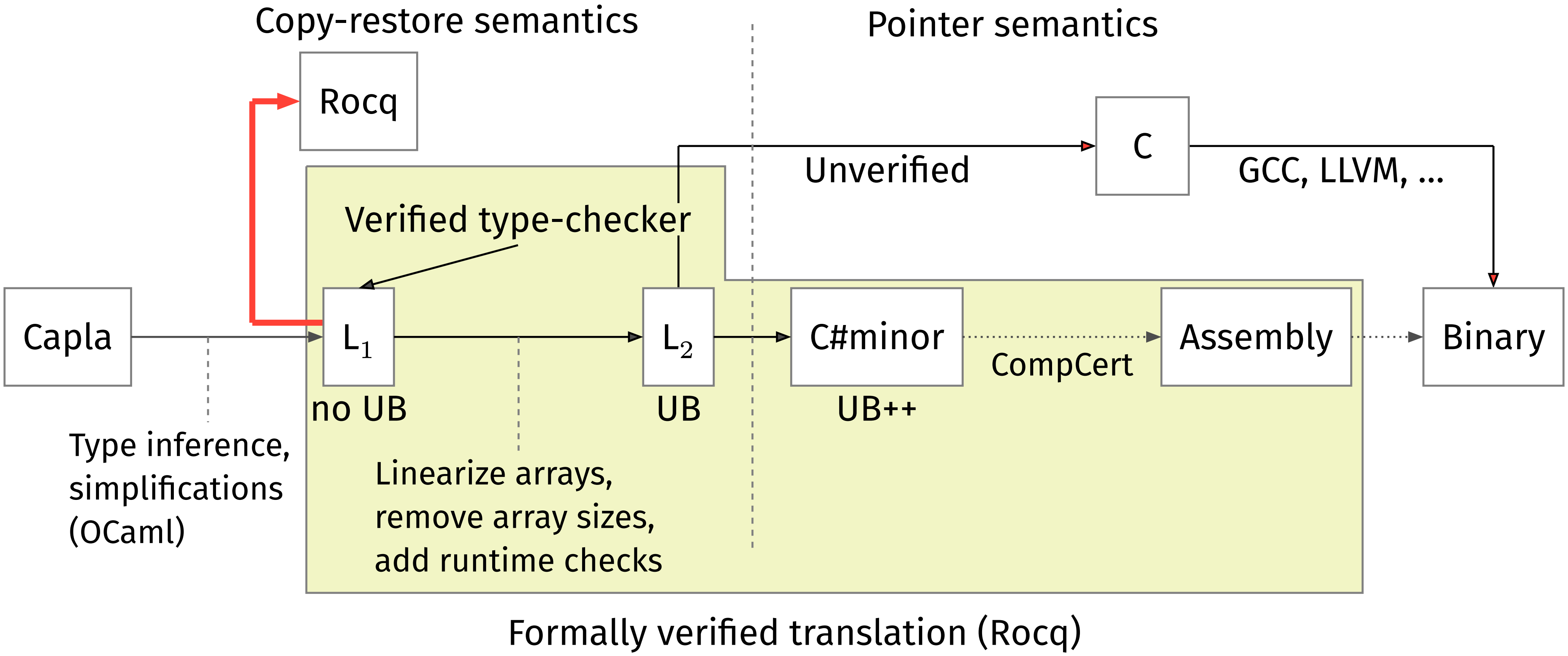
Part 2: the Capla compiler

- Translates Capla code to assembly
  - Using CompCert as a backend
  - Efficient implementation of the copy-restore semantics
  - Interoperability between both Capla and C code
- Correctness proof for both the compiler and the type-checker
  - Formally verified with Rocq
    Code: $\sim$10,000 loc, spec: $\sim$11,000 loc, proof: $\sim$17,000 loc
  - Most difficult part: Relation between the memory models of L_2 and C#minor

Part 2: the Capla compiler

- Translates Capla code to assembly
  - Using CompCert as a backend
  - Efficient implementation of the copy-restore semantics
  - Interoperability between both Capla and C code
- Correctness proof for both the compiler and the type-checker
  - Formally verified with Rocq
    Code: $\sim$10,000 loc, spec: $\sim$11,000 loc, proof: $\sim$17,000 loc
  - Most difficult part: Relation between the memory models of L_2 and C#minor
- Unverified C backend for performance

Future work: verifying and running Capla code inside of Rocq



Future work: verifying and running Capla code inside of Rocq

Application: Turn Rocq into a computer algebra system

- Invoke Capla functions from Rocq
- Prove correctness theorems to use their results

```
Theorem is_prime_correct n b:
  eval is_prime_capla [Vint64 n] (Some (Vbool b)) -> is_prime n = b.
Proof. (* complicated proof using the semantics of L1 *) Qed.

Axiom is_prime_exec n v:
  exec is_prime_compiled n = Some v -> eval is_prime_capla [Vint64 n] (Some v).

Goal is_prime 298348787309 = true.
Proof.
  apply is_prime_correct. (* The Capla code is correct *)
  apply is_prime_exec. (* The Capla code was correctly compiled *)
  now compute. (* The Capla code is executed inside Rocq's kernel *)
Qed.
```

Appendix

Views

```
  fun karatsuba(r: mut [i64; 2 * n],
                a b:   [i64; n],
                t: mut [i64; 2 * n], n: 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);
      sub2(t0, r0, 2 * k);
      sub2(t0, r2, 2 * k);
    }
    add2(r[k..(3 * k)], t0, 2 * k);
  }
```

$k \mapsto 2$

$a \mapsto \boxed{1}\boxed{2}\boxed{3}\boxed{4}$

$b \mapsto \boxed{6}\boxed{4}\boxed{2}\boxed{0}$

$t \mapsto \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$

$r \mapsto \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$

Views

```
  fun karatsuba(r: mut [i64; 2 * n],
                a b:    [i64; n],
                t: mut [i64; 2 * n], n: 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);
      sub2(t0, r0, 2 * k);
      sub2(t0, r2, 2 * k);
    }
    add2(r[k..(3 * k)], t0, 2 * k);
  }
```

$k \mapsto 2$

$a_0 \mapsto \begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array}$ $\quad a_1 \mapsto \begin{array}{|c|c|}\hline 3 & 4 \\ \hline \end{array}$ $\quad a \mapsto \emptyset$

$b_0 \mapsto \begin{array}{|c|c|}\hline 6 & 4 \\ \hline \end{array}$ $\quad b_1 \mapsto \begin{array}{|c|c|}\hline 2 & 0 \\ \hline \end{array}$ $\quad b \mapsto \emptyset$

$t_0 \mapsto \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline \end{array}$ $\quad t_1 \mapsto \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline \end{array}$ $\quad t \mapsto \emptyset$

$r_0 \mapsto \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline \end{array}$ $\quad r_2 \mapsto \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline \end{array}$ $\quad r \mapsto \emptyset$

Views

```
fun karatsuba(r: mut [i64; 2 * n],
              a b:   [i64; n],
              t: mut [i64; 2 * n], n: 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);
    sub2(t0, r0, 2 * k);
    sub2(t0, r2, 2 * k);
→
  }
  add2(r[k..(3 * k)], t0, 2 * k);
}
```

$k \mapsto 2$

$a_0 \mapsto \boxed{1}\boxed{2}$ $a_1 \mapsto \boxed{3}\boxed{4}$ $a \mapsto \emptyset$

$b_0 \mapsto \boxed{6}\boxed{4}$ $b_1 \mapsto \boxed{2}\boxed{0}$ $b \mapsto \emptyset$

$t_0 \mapsto \boxed{20}\boxed{40}\boxed{16}\boxed{0}$ $t_1 \mapsto \dots$ $t \mapsto \emptyset$

$r_0 \mapsto \boxed{6}\boxed{16}\boxed{8}\boxed{0}$ $r_2 \mapsto \boxed{6}\boxed{8}\boxed{0}\boxed{0}$ $r \mapsto \emptyset$

Views

```
fun karatsuba(r: mut [i64; 2 * n],
              a b:   [i64; n],
              t: mut [i64; 2 * n], n: 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);
    sub2(t0, r0, 2 * k);
    sub2(t0, r2, 2 * k);
  }
→ add2(r[k..(3 * k)], t0, 2 * k);
}
```

$$k \mapsto 2$$

$$a_0 \mapsto \begin{array}{|c|c|}\hline 1 & 2 \\ \hline\end{array} \qquad a_1 \mapsto \begin{array}{|c|c|}\hline 3 & 4 \\ \hline\end{array} \qquad a \mapsto \emptyset$$

$$b_0 \mapsto \begin{array}{|c|c|}\hline 6 & 4 \\ \hline\end{array} \qquad b_1 \mapsto \begin{array}{|c|c|}\hline 2 & 0 \\ \hline\end{array} \qquad b \mapsto \emptyset$$

$$t_0 \mapsto \begin{array}{|c|c|c|c|}\hline 20 & 40 & 16 & 0 \\ \hline\end{array} \qquad t_1 \mapsto \ldots \qquad t \mapsto \emptyset$$

$$r \mapsto \begin{array}{|c|c|c|c|c|c|c|c|}\hline 6 & 16 & 8 & 0 & 6 & 8 & 0 & 0 \\ \hline\end{array}$$

Appendix

Reimplementing BLAS functions

Capla

```
fun b_sgemv_n(
  trans: u8, m n: i32, alpha: f32,
  a:     [f32; (u64) n, (u64) lda], lda: i32,
  x:     [f32; (u64) (1 + (n - 1) * incx)], incx: i32,
  beta: f32,
  y: mut [f32; (u64) (1 + (m - 1) * incy)], incy: i32) {
  ...
  jx = kx;
  if incy == 1 {
    for j: u32 = 0 .. (u32) n {
      temp = alpha * x[jx];
      for i: u32 = 0 .. (u32) m {
        y[i] = y[i] + temp * a[j, i];
      }
      jx = jx + incx;
    }
  } else {
    for j: u32 = 0 .. (u32) n {
      temp = alpha * x[jx];
      iy = ky;
      for i: u32 = 0 .. (u32) m {
        y[iy] = y[iy] + temp * a[j, i];
        iy = iy + incy;
      }
      jx = jx + incx;
    }
    ...
```

Fortran

```
SUBROUTINE SGEMV(TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
      REAL ALPHA,BETA
      INTEGER INCX,INCY,LDA,M,N
      CHARACTER TRANS
      REAL A(LDA,*),X(*),Y(*)
      ...
      IF (LSAME(TRANS,'N')) THEN
          JX = KX
          IF (INCY.EQ.1) THEN
              DO 60 J = 1,N
                  TEMP = ALPHA*X(JX)
                  DO 50 I = 1,M
                      Y(I) = Y(I) + TEMP*A(I,J)
  50              CONTINUE
                  JX = JX + INCX
  60          CONTINUE
          ELSE
              DO 80 J = 1,N
                  TEMP = ALPHA*X(JX)
                  IY = KY
                  DO 70 I = 1,M
                      Y(IY) = Y(IY) + TEMP*A(I,J)
                      IY = IY + INCY
  70              CONTINUE
                  JX = JX + INCX
  80          CONTINUE
              ...
```

Reimplementing BLAS functions

```
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 {
    assert (1 + (n - 1) * incx == n);
    assert (1 + (n - 1) * incy == n);
    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 {
    ...
  } }
```

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

Eliminated but the compiler needs a bit of help

Dynamic test: $1 < 2$

Trivially eliminated

Straightforward translation from the original BLAS implementation in Fortran

Function call and return

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

$$\dfrac{}{(E_{\texttt{f}}, S_{\texttt{f}}, \langle \texttt{return}\ v\rangle, \texttt{Kcall}(y, E', S, m, k)) \to (E'[a_i \leftarrow E_{\texttt{f}}(x_i), ...][y \leftarrow v], S, \langle\rangle, k)}\ \textsc{Return}$$

🟨 Statically guaranted by typing

🟥 Dynamically tested

Function call and return

		CompCert
BLAS	saxpy	4.58
	zdotu	2.26
	sgemv	1.71
	dgemv	2.86
	dtrsv (N)	2.07
	dtrsv (T)	2.10
GMP	mpn_add_n	4.77
	mpn_addmul_1	2.83
	mpn_mul (+ its deps)	3.26
	mpn_mul (only)	1.19

- ≥ 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 14.3.0, optimization level `-O2 -ftree-vectorize`

Function call and return

		CompCert	GCC	LLVM
BLAS	`saxpy`	4.58	3.52	0.76
	`zdotu`	2.26	2.25	1.16
	`sgemv`	1.71	1.14	0.31
	`dgemv`	2.86	1.26	0.89
	`dtrsv` (N)	2.07	1.12	1.37
	`dtrsv` (T)	2.10	1.49	0.89
GMP	`mpn_add_n`	4.77	2.33	2.70
	`mpn_addmul_1`	2.83	1.20	1.07
	`mpn_mul` (+ its deps)	3.26	1.58	1.57
	`mpn_mul` (only)	1.19	1.02	1.03

- ≥ 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 14.3.0, optimization level `-O2 -ftree-vectorize`

Function call and return

		CompCert	GCC	LLVM	GCC with assertions	LLVM with assertions
BLAS	`saxpy`	4.58	3.52	0.76	3.23	0.74
	`zdotu`	2.26	2.25	1.16	1.01	1.15
	`sgemv`	1.71	1.14	0.31	1.16	0.31
	`dgemv`	2.86	1.26	0.89	1.81	0.88
	`dtrsv` (N)	2.07	1.12	1.37	0.90	1.19
	`dtrsv` (T)	2.10	1.49	0.89	0.92	0.89
GMP	`mpn_add_n`	4.77	2.33	2.70	–	–
	`mpn_addmul_1`	2.83	1.20	1.07	–	–
	`mpn_mul` (+ its deps)	3.26	1.58	1.57	–	–
	`mpn_mul` (only)	1.19	1.02	1.03	–	–

- ≥ 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 14.3.0, optimization level `-O2 -ftree-vectorize`

Example: Schoolbook multiplication of polynomials in F_2

```
fun addmul_1(rp: mut [u8; n], up: [u8; n], n: u64, vl: u8) {
  for i = 0 .. n {
    rp[i] = (u8) (rp[i] ^ (u8) (up[i] & vl));
  }
}

fun mul_1(rp: mut [u8; n], up: [u8; n], n: u64, vl: u8) {
  for i = 0 .. n {
    rp[i] = (u8) (up[i] & vl);
  }
}

fun mul(rp: mut [u8; un + vn], up: [u8; un], un: u64, vp: [u8; vn], vn: u64) {
  mul_1(rp[..un], up, un, vp[0]);
  for i = 1 .. vn {
    addmul_1(rp[i..(un + i)], up, un, vp[i]);
  }
}
```

Example: Schoolbook multiplication of polynomials in F_2

```
Theorem mul_correct:
  ∀ u nu v nv e1 result,
    ...
    eval_funcall mul
                 [Varr r; Varr u; Vint64 nu; Varr v; Vint64 nv]
                 e1 (Some result) ->
    ∃ lv,
      e1!(param 0 mul) = Some (Varr lv) /\
      ...
      (to_poly lv (nnu + nnv) = to_poly u nnu * to_poly v nnv)%R.
```

Example: Schoolbook multiplication of polynomials in F_2

On successful execution of `mul`

```
Theorem mul_correct:
  ∀ u nu v nv e1 result,
    ...
    eval_funcall mul
                 [Varr r; Varr u; Vint64 nu; Varr v; Vint64 nv]
                 e1 (Some result) ->
    ∃ lv,
      e1!(param 0 mul) = Some (Varr lv) /\
      ...
      (to_poly lv (nnu + nnv) = to_poly u nnu * to_poly v nnv)%R.
```

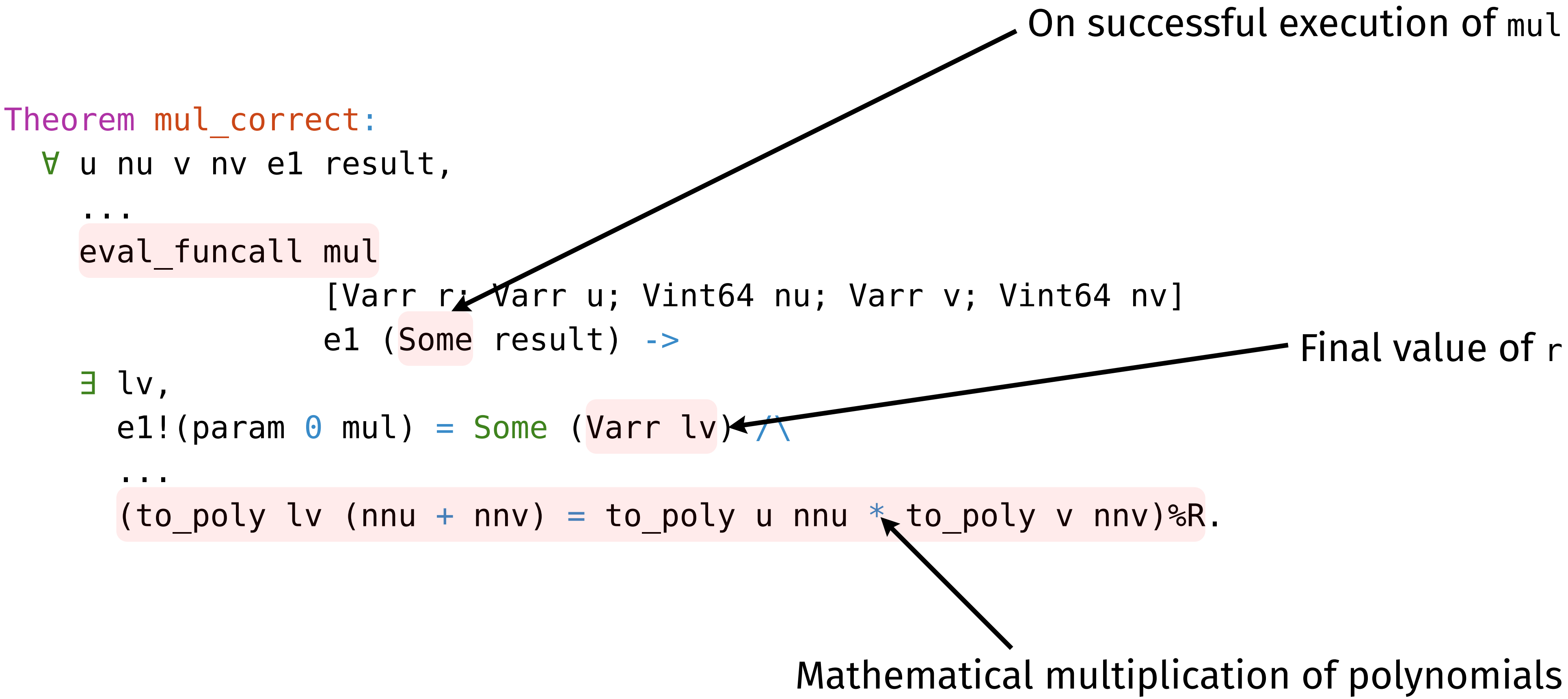
Example: Schoolbook multiplication of polynomials in F_2

On successful execution of `mul`

```
Theorem mul_correct:
  ∀ u nu v nv e1 result,
    ...
    eval_funcall mul
                 [Varr r; Varr u; Vint64 nu; Varr v; Vint64 nv]
                 e1 (Some result) ->
    ∃ lv,
      e1!(param 0 mul) = Some (Varr lv) /\
      ...
      (to_poly lv (nnu + nnv) = to_poly u nnu * to_poly v nnv)%R.
```

Final value of `r`

Example: Schoolbook multiplication of polynomials in F_2



```
Theorem mul_correct:
  ∀ u nu v nv e1 result,
    ...
    eval_funcall mul
                 [Varr r; Varr u; Vint64 nu; Varr v; Vint64 nv]
                 e1 (Some result) ->
    ∃ lv,
      e1!(param 0 mul) = Some (Varr lv) /\
      ...
      (to_poly lv (nnu + nnv) = to_poly u nnu * to_poly v nnv)%R.
```

On successful execution of `mul`

Final value of `r`

Mathematical multiplication of polynomials