# A language for computer algebra and its formally verified compiler

Non-permanent LMF seminar

January 30, 2024

Josué Moreau

# Computer Algebra

- Algorithms working with mathematical objects (matrices, polynomials, etc)
- Efficiency $\Rightarrow$ Specialized libraries: BLAS (linear algebra), GMP (multi-precision integers), etc

# Bugs

**Example (GMP $\leq$ 5.1.1)**

`mpz_powm_ui(r, b, e, m)`: $\mathtt{r} \leftarrow \mathtt{b}^{\mathtt{e}} \mod \mathtt{m}$

Computes garbage if `b` is over 15000 decimal.

$\Rightarrow$ We want to verify computer algebra programs.

# Bugs

**Example (GMP $\leq$ 5.1.1)**

`mpz_pown_ui(r, b, e, m)`: $\texttt{r} \leftarrow \texttt{b}^{\texttt{e}} \mod \texttt{m}$

Computes garbage if `b` is over 15000 decimal.

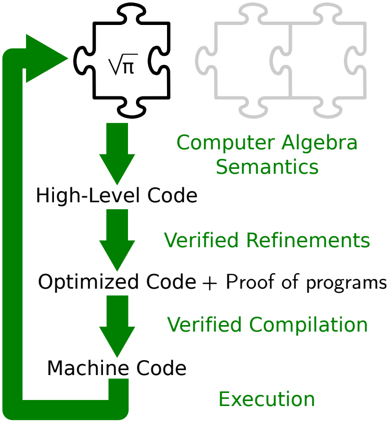
$\Rightarrow$ We want to verify computer algebra programs.

**Example (GMP 6.2.0)**

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

$\Rightarrow$ We have to be sure compilers don't introduce bugs.

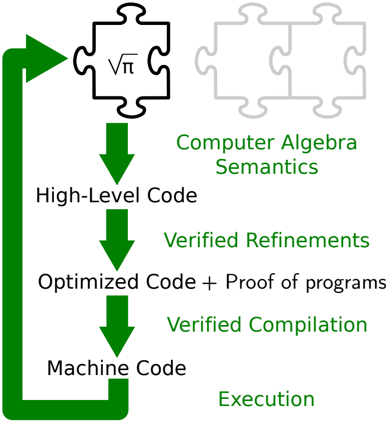
# FRESCO: Fast and Reliable Symbolic Computation




Turn the Coq proof assistant
into an environment where

- fast implementations of computer algebra algorithms can be written and verified
- machine code will be executed in Coq
- results will be used in proofs

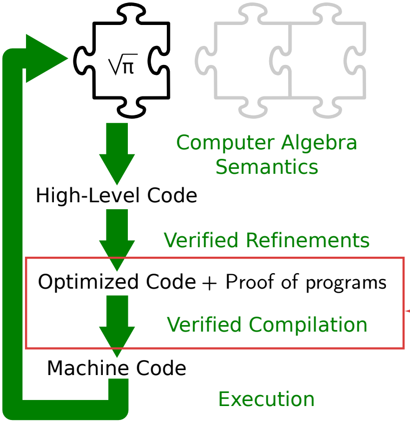
# FRESCO: Fast and Reliable Symbolic Computation




Turn the Coq proof assistant
into an environment where

- fast implementations
  of computer algebra algorithms
  can be written and verified
- machine code will be executed in Coq
- results will be used in proofs

# FRESCO: Fast and Reliable Symbolic Computation




Turn the Coq proof assistant into an environment where

- fast implementations of computer algebra algorithms can be written and verified
- machine code will be executed in Coq
- results will be used in proofs

My work

# Goals

- a low-level language:
  - suitable for computer algebra algorithms (e.g., arrays, matrices)
  - safe (e.g., no access outside the memory of the program)
  - with constructions simplifying the proof of programs (e.g., no aliasing)
- a formally verified compiler for this language (such as CompCert)

# Some existing approaches

Rust : safe language

+ many interesting constructions
- but the compiler is not verified

# Some existing approaches

Rust : safe language

+ many interesting constructions
- but the compiler is not verified

CakeML (SML) : safe language

+ formally verified compiler
- but poorly suited for libraries such as GMP and BLAS

# Some existing approaches

Rust : safe language

- \+ many interesting constructions
- \- but the compiler is not verified

CakeML (SML) : safe language

- \+ formally verified compiler
- \- but poorly suited for libraries such as GMP and BLAS

VST/CompCert (C) : unsafe language

- \+ user can **prove** safety and correctness
- \+ formally verified compiler (CompCert)

# Some existing approaches

Rust : safe language

+ many interesting constructions
- but the compiler is not verified

CakeML (SML) : safe language

+ formally verified compiler
- but poorly suited for libraries such as GMP and BLAS

VST/CompCert (C) : unsafe language

+ user can **prove** safety and correctness
+ formally verified compiler (CompCert)

Why3 (WhyML) and frama-C (C) : unsafe languages

+ user can **prove** safety and correctness more automatically (ex: WhyMP)
- but the compiler (at least extraction) is not proved for Why3
- even if we compile a verified ACSL program with CompCert,
  no guarantee that C semantics of CompCert and frama-C agree

# Table of contents


1 Design of the language

2 Semantics

3 Compilation

# Design of the language

# Matrix Multiplication

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

# Matrix Multiplication

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

Array size passed explicitly as arguments

# Matrix Multiplication

Same size for multiple arrays

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

Array size passed explicitly as arguments

# Matrix Multiplication

Same size for multiple arrays

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

Array size passed explicitly as arguments

Distinction mutable/persistent arrays
+ borrowing
⇒ proof of program

# Paths and expressions

Syntactic path:

$$q \quad ::= \quad id[\vec{e}][\vec{e}]\ldots$$

Expressions:

$$\begin{array}{rcll}
e & ::= & c & \text{constants} \\
& | & (\tau_1 \to \tau_2)e & \text{cast} \\
& | & op_1(e) & \text{unary operations } (\texttt{not}, \texttt{neg}) \\
& | & op_2(e_1, e_2) & \text{binary operations } (+, -, *, /, \gg, \ldots) \\
& | & q & \text{read}
\end{array}$$

# Instructions

$$
\begin{array}{rcll}
s & ::= & \texttt{skip} & \\
& | & q \leftarrow e & \text{writing} \\
& | & id^{?} \leftarrow f(q_1, \ldots, q_n) & \text{function call} \\
& | & s_1 ; s_2 & \text{sequence} \\
& | & \texttt{return}\ e^{?} & \\
& | & \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\} & \\
& | & \texttt{loop}\ \{s\} & \\
& | & \texttt{break} \mid \texttt{continue} & \\
& | & \texttt{error} &
\end{array}
$$

# Functions

$$sig ::= \{\texttt{args} = \vec{\tau}; \texttt{res} = \tau\}$$

$$
\begin{array}{lll}
\mathcal{F} & ::= \{ & \\
& \quad \texttt{sig} & = sig \\
& \quad \texttt{params} & = \vec{id} \\
& \quad \texttt{vars} & = \vec{id} \\
& \quad \texttt{tenv} & = id \rightharpoonup \tau \qquad \text{(written } \Gamma_F \text{ in next slides)} \\
& \quad \texttt{szenv} & = id \rightharpoonup [[\vec{e}], \ldots] \qquad \text{(written } \Sigma_F \text{ in next slides)} \\
& \quad \texttt{penv} & = id \rightharpoonup \{\texttt{Shared}, \texttt{Mut}, \texttt{Own}\} \qquad \text{(written } \rho_F \text{ in next slides)} \\
& \quad \texttt{body} & = s \\
& \} &
\end{array}
$$

And some properties on functions (e.g. $\forall x, \forall s \in \Sigma_F(x), \rho_F(s) = \texttt{Shared}$).

# Example: Multiplication of polynomials

```
fun mul_poly(a: [i64; m], b: [i64; n], dest: mut [i64; m + n - 1],
             m n: u64) {
    for i: u64 = 0 .. m
        for j: u64 = 0 .. n
            dest[i + j] = dest[i + j] + a[i] * b[j];
}
```

# Example: Multiplication of polynomials

```
fun mul_poly(a: [i64; m], b: [i64; n], dest: mut [i64; m + n - 1],
             m n: u64) {
    for i: u64 = 0 .. m
        for j: u64 = 0 .. n
            dest[i + j] = dest[i + j] + a[i] * b[j];
}
```

```
fun mul_poly(a: [i64; da + 1], b: [i64; db + 1],
             dest: mut [i64; da + db + 1],
             da db: u64) {
    for i: u64 = 0 .. (da + 1)
        for j: u64 = 0 .. (db + 1)
            dest[i + j] = dest[i + j] + a[i] * b[j];
}
```

# Semantics

# Semantics: operations and errors

$$\text{Edivs}\,\frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \qquad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \qquad i_2 \neq 0 \qquad i_1 \neq \texttt{min\_sint} \lor i_2 \neq -1}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{Vint}\ (i_1 / i_2)}$$

# Semantics: operations and errors

$$\text{Edivs}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \qquad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \qquad i_2 \neq 0 \qquad i_1 \neq \texttt{min\_sint} \vee i_2 \neq -1}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{Vint}\ (i_1 / i_2)}$$

$$\text{EdivsErr}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \qquad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \qquad i_2 = 0 \vee (i_1 = \texttt{min\_sint} \wedge i_2 = -1)}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{error}}$$

# Semantics: operations and errors

$$\text{Edivs}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \qquad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \qquad i_2 \neq 0 \qquad i_1 \neq \texttt{min\_sint} \lor i_2 \neq -1}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{Vint}\ (i_1 / i_2)}$$

$$\text{EdivsErr}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \qquad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \qquad i_2 = 0 \lor (i_1 = \texttt{min\_sint} \land i_2 = -1)}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{error}}$$

$$\text{EdivsErr1}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{error} \qquad E, F \vdash e_2 \Rightarrow v / \texttt{error}}{E, F \vdash \mathsf{divs}(e_1, e_2) \Rightarrow \texttt{error}}$$

# Semantics: casts

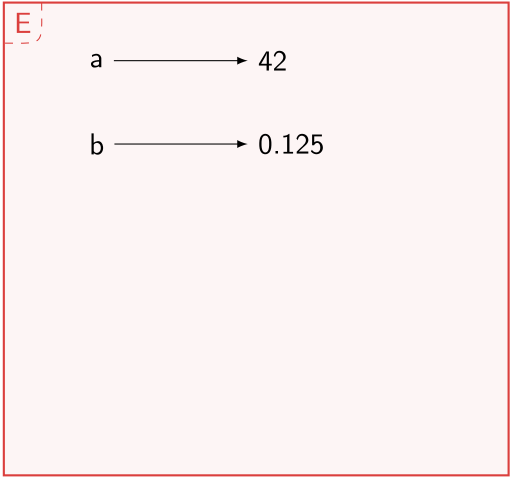
$$\text{EcastSIntF64}\ \frac{E, F \vdash e \Rightarrow \texttt{Vint}\ n}{E, F \vdash (\texttt{int}_{32,\texttt{Signed}} \rightarrow \texttt{float}_{64})e \Rightarrow \texttt{Vfloat}_{64}\ f_n}$$

$$\text{EcastF64SInt64}\ \frac{\begin{array}{c} E, F \vdash e \Rightarrow \texttt{Vfloat}_{64}\ f \\ -2^{63} \leq f < 2^{63} \end{array}}{E, F \vdash (\texttt{float}_{64} \rightarrow \texttt{int}_{64,Signed})e \Rightarrow \texttt{Vint}_{64}\ n_f}$$

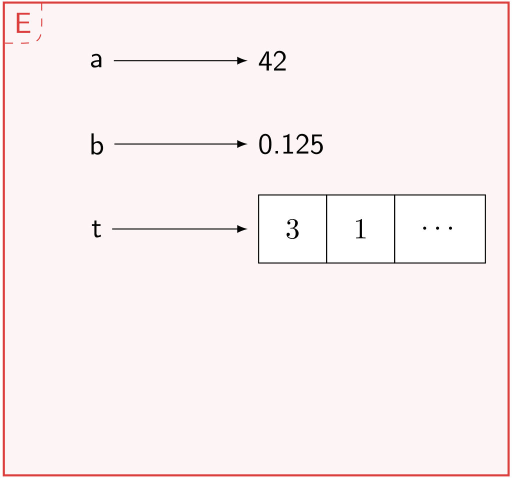
$$\text{EcastF64SInt64Err}\ \frac{\begin{array}{c} E, F \vdash e \Rightarrow \texttt{Vfloat}_{64}\ f \\ f < -2^{63} \vee f \geq 2^{63} \end{array}}{E, F \vdash (\texttt{float}_{64} \rightarrow \texttt{int}_{64,Signed})e \Rightarrow \texttt{error}}$$

# Memory Model

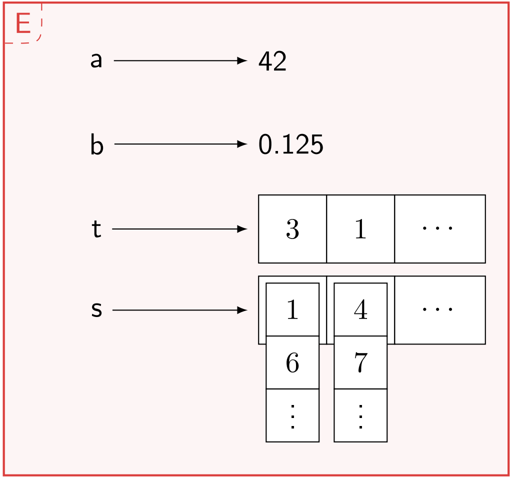
# Memory Model



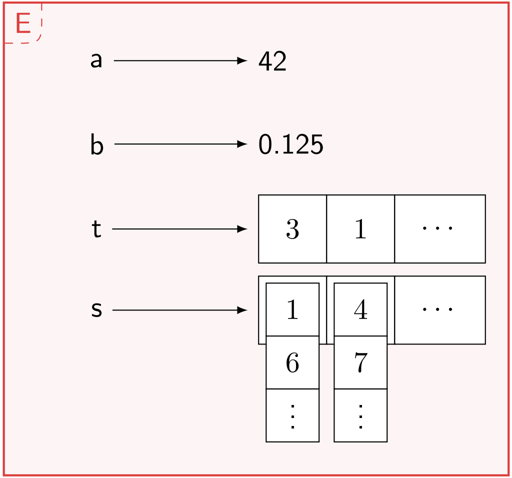
# Memory Model



# Memory Model



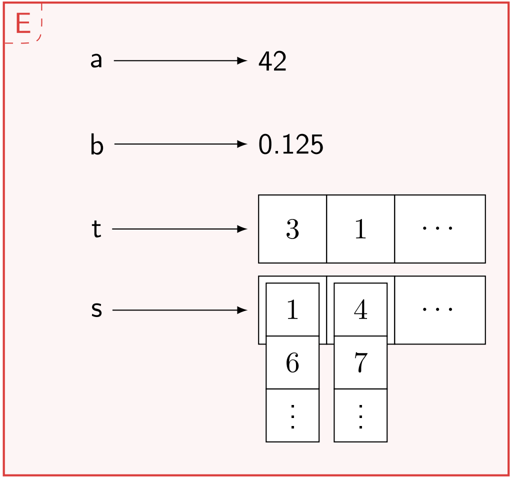
# Memory Model




Semantic paths:

$$p \quad ::= \quad id[n][n]\ldots \qquad \text{(linear array)}$$

# Memory Model

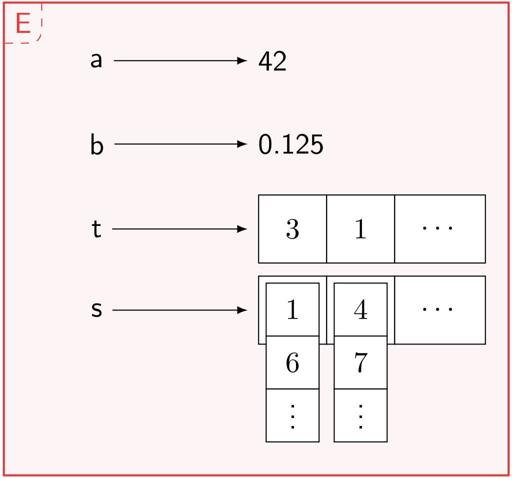



Semantic paths:

$$p \quad ::= \quad id[n][n]\ldots \qquad \text{(linear array)}$$

$\texttt{dest}[i, j][k] \Rightarrow$
$\texttt{dest}[i * \texttt{sizeof(dest[0])} + j][k]$

# Memory Model




Semantic paths:

$p \quad ::= \quad id[n][n]\ldots \qquad$ (linear array)

$\texttt{dest}[i, j][k] \Rightarrow$
$\texttt{dest}[i * \texttt{sizeof(dest[0])} + j][k]$

$$E[(a, [])] = 42$$
$$E[(b, [])] = 0.125$$
$$E[(t, [0])] = 3$$
$$E[(s, [1; 1])] = 7$$

# Semantics: instructions

Step in the semantics ($G$ is the definition of all functions):

$$G \vdash st \to st'$$

$$\begin{array}{rcll} st & ::= & \mathcal{S}(E, F, s, k) & \text{regular state} \\ & \mid & \mathcal{C}(F, \vec{v}, k) & \text{call state} \\ & \mid & \mathcal{R}(E, F, v, k) & \text{return state} \end{array}$$

$$\begin{array}{rcll} k & ::= & \texttt{Kstop} & \text{end of program} \\ & \mid & \texttt{Kseq}(s, k) & \text{sequence} \\ & \mid & \texttt{Kloop}(s, k) & \text{loop} \\ & \mid & \texttt{Kreturnto}(id^?, E, F, m, k) & \text{return} \end{array}$$

# Semantics: writing

$$\text{Write}\ \frac{E, F \vdash q \Rightarrow p \qquad p = (i, \vec{z}) \qquad E, F \vdash e \Rightarrow v \qquad \texttt{primitive\_value}(v) \qquad \Gamma_F(p) = \tau \qquad v \in \tau \qquad P_F(i) \geq \texttt{Mut}}{\mathcal{S}(E, F, q \leftarrow e, k) \rightarrow \mathcal{S}(E[p \mapsto v], F, \texttt{skip}, k)}$$

$$\text{WriteErr}\ \frac{E, F \vdash q \Rightarrow \texttt{error}}{\mathcal{S}(E, F, q \leftarrow e, k) \rightarrow \mathcal{S}(E, F, \texttt{error}, k)}$$

# Semantics: function call

$$\dfrac{\begin{array}{c} G(id_f) = \texttt{Internal}(F') \qquad |\vec{a}| = |F'.\texttt{sig.sig\_args}| \\ E, F \vdash \vec{a} \Rightarrow \vec{p} \qquad E(\vec{p}) = \vec{v} \qquad \vec{v} \in F'.\texttt{sig.sig\_args} \\ \forall i, P_F(p_i) \geq P_{F'}(F.\texttt{params}_i) \\ \forall i, \Gamma_F(p_i) = \Gamma_{F'}(F'.\texttt{params}_i) \\ \texttt{valid\_call}(E, F, \texttt{Internal}(F'), p) \\ \forall i\ j, i \neq j \land P_{F'}(F'.\texttt{params}_i) \geq \texttt{Mut} \rightarrow p_i \not\preceq p_j \land p_j \not\preceq p_i \end{array}}{\begin{array}{l} \mathcal{S}(E, F, \texttt{call}(id_v, id_f, \vec{a}), k) \rightarrow \mathcal{C}(F', \vec{x}, \texttt{Kreturnto}(id_v, E, F, m, k)) \\ \forall i, x_i = (p_i, v_i) \qquad m = \{(p_i, F'.\texttt{params}_i) \mid P_{F'}(F'.\texttt{params}_i) \geq \texttt{Mut}\} \end{array}}\ \text{CallInternal}$$

# Semantics: return

$$\frac{\begin{array}{c}\forall (p, i) \in m, E[p] = \texttt{Varr \_} \ \land \ E'[i] = \texttt{Varr \_} \\ \forall (p, i) \in m, \Gamma_F[p] = \Gamma_{F'}[i] \\ \texttt{primitive\_value}(v) \\ E_{upd} = \texttt{update\_env}(E, m, E')\end{array}}{\mathcal{R}(E', F', v, \texttt{Kreturnto}(id_v, E, F, m, k) \to \mathcal{S}(E_{upd}[id_v \mapsto v], F, \texttt{skip}, k)}\ \text{Return}$$

# Proof of program

- Environments trivially express the absence of alias
- Anything which is not passed (as mutable) to a called function is not modified
- Multidimensional arrays avoid using non linear arithmetic
- Easy WP computation

# Compilation

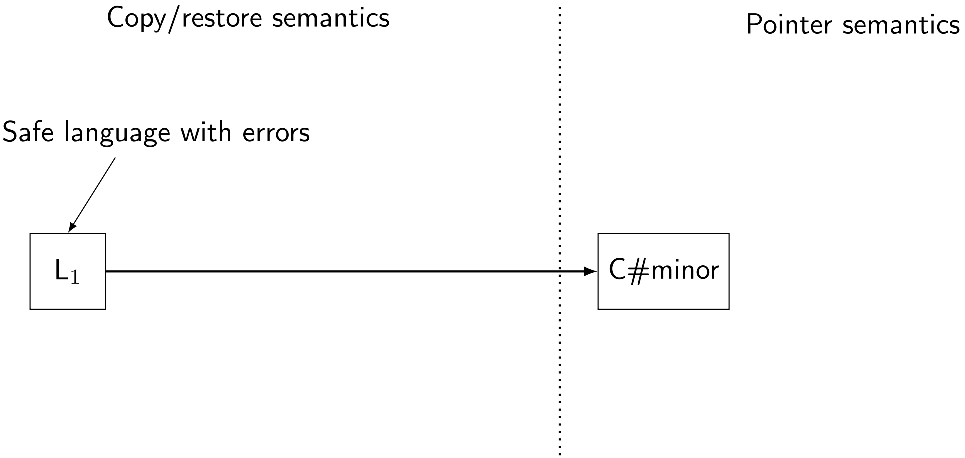
# Compilation

Copy/restore semantics

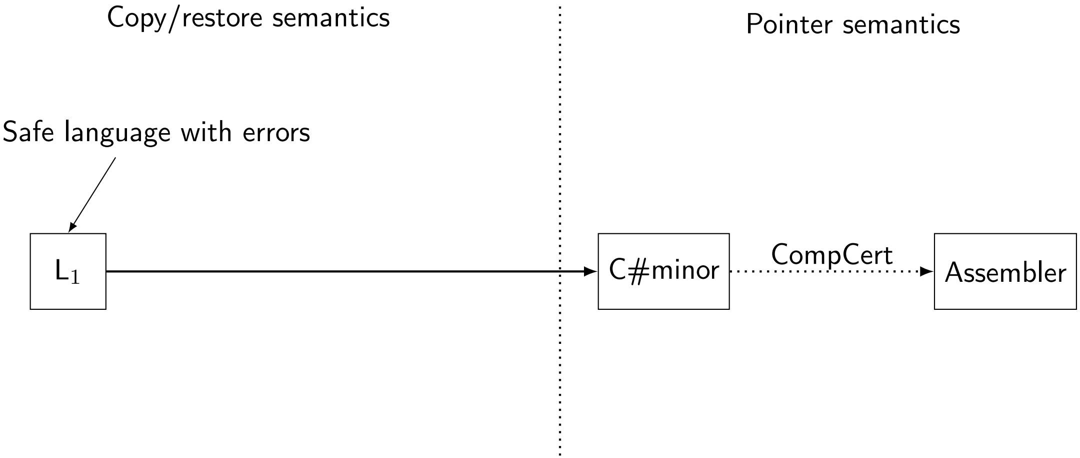
Safe language with errors

$\boxed{L_1}$

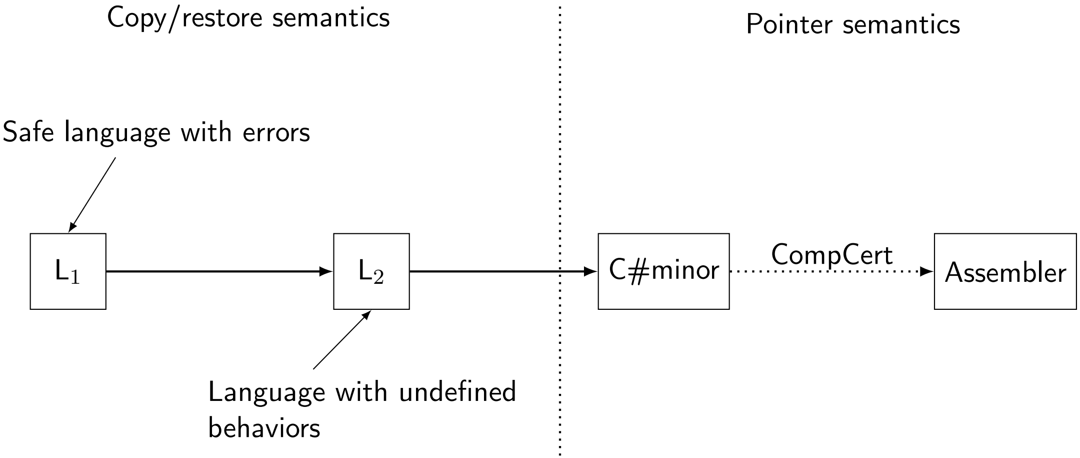
# Compilation



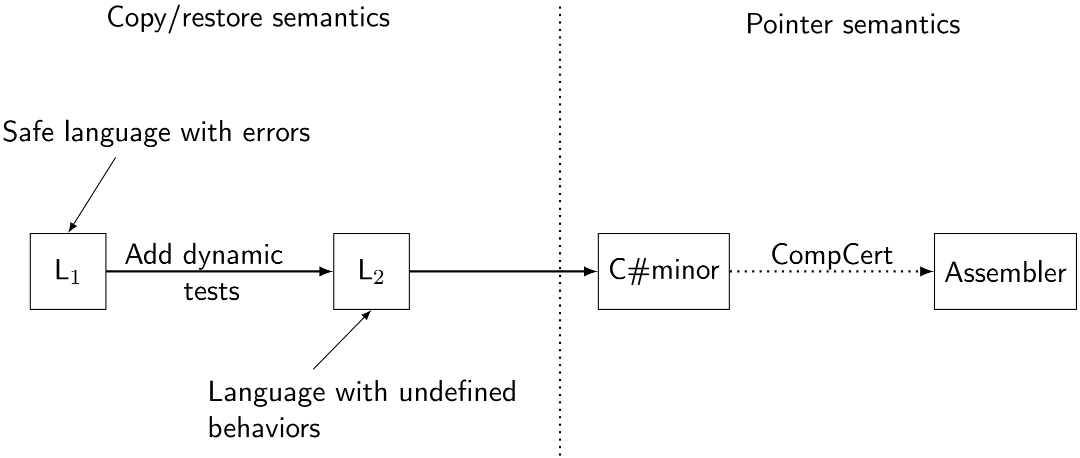
# Compilation



# Compilation



# Compilation



# Translation $L_1 \to L_2$: test generation

$$\texttt{ET}\big(\texttt{divu}(e_1, e_2)\big) \quad = \quad \texttt{ET}(e_1) +\!\!+\ \texttt{ET}(e_2) +\!\!+\ (e_2 \neq 0)$$

The order of tests is important.

# Translation $\mathsf{L}_1 \to \mathsf{L}_2$: test generation

$$
\begin{aligned}
\mathtt{ET}(\mathtt{divu}(e_1, e_2)) &= \mathtt{ET}(e_1) +\!\!+\ \mathtt{ET}(e_2) +\!\!+\ (e_2 \neq 0) \\
\mathtt{ET}((\mathtt{float}_{64} \to \mathtt{int}_{32,\mathtt{Signed}})e) &= \mathtt{ET}(e) +\!\!+\ (-2^{31} - 1 < e < 2^{31})
\end{aligned}
$$

The order of tests is important.

# Translation $L_1 \to L_2$: test generation

$$
\begin{aligned}
\mathtt{ET}(\mathtt{divu}(e_1, e_2)) &= \mathtt{ET}(e_1) +\!\!+\ \mathtt{ET}(e_2) +\!\!+\ (e_2 \neq 0) \\
\mathtt{ET}((\mathtt{float}_{64} \to \mathtt{int}_{32,\mathtt{Signed}})e) &= \mathtt{ET}(e) +\!\!+\ (-2^{31} - 1 < e < 2^{31}) \\
\mathtt{ET}((\mathtt{float}_{32} \to \mathtt{int}_{32,\mathtt{Unsigned}})e) &= \mathtt{ET}(e) +\!\!+\ (-1 < e < 2^{32})
\end{aligned}
$$

The order of tests is important.

# Translation $\mathsf{L}_1 \to \mathsf{L}_2$: test generation

$$
\begin{array}{lcl}
\mathtt{ET}(\mathtt{divu}(e_1, e_2)) & = & \mathtt{ET}(e_1) \mathbin{+\!\!+} \mathtt{ET}(e_2) \mathbin{+\!\!+} (e_2 \neq 0) \\
\mathtt{ET}((\mathtt{float}_{64} \to \mathtt{int}_{32,\mathtt{Signed}})e) & = & \mathtt{ET}(e) \mathbin{+\!\!+} (-2^{31} - 1 < e < 2^{31}) \\
\mathtt{ET}((\mathtt{float}_{32} \to \mathtt{int}_{32,\mathtt{Unsigned}})e) & = & \mathtt{ET}(e) \mathbin{+\!\!+} (-1 < e < 2^{32}) \\
\mathtt{ET}(x[i_1, \ldots, i_k]) & = & \mathtt{ET}(i_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathtt{ET}(i_k) \mathbin{+\!\!+} \\
 & & (i_1 <_u s_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} (i_k <_u s_k) \\
 & & \text{where } s_1, \ldots, s_k \text{ are the size variables of } x
\end{array}
$$

The order of tests is important.

# Translation $L_2 \to$ C#minor

Translation from $L_2$ to C#minor is mostly a 1-to-1 translation, except for the following constructions:

$$
\begin{aligned}
\mathtt{TrExp}(id[e_1, \ldots, e_k]) \quad = \quad & *\Big(id^t + \mathtt{sizeof}(id[0, \ldots, 0]) \times \\
& \quad \big((((e_1^t \times s_2^t + e_2^t) \times s_3^t + \ldots) \ldots) \times s_k^t + e_k^t\big)\Big) \\
& \text{where } e^t = \mathtt{TrExp}(e)
\end{aligned}
$$

# Translation $\mathrm{L}_2 \to$ C#minor

Translation from $\mathrm{L}_2$ to C#minor is mostly a 1-to-1 translation, except for the following constructions:

$$
\begin{aligned}
\texttt{TrExp}(id[e_1, \ldots, e_k]) \quad &= \quad *\Big(id^t + \texttt{sizeof}(id[0, \ldots, 0])\times \\
&\qquad \big((((e_1^t \times s_2^t + e_2^t) \times s_3^t + \ldots) \ldots) \times s_k^t + e_k^t\big)\Big) \\
&\quad \text{where } e^t = \texttt{TrExp}(e)
\end{aligned}
$$

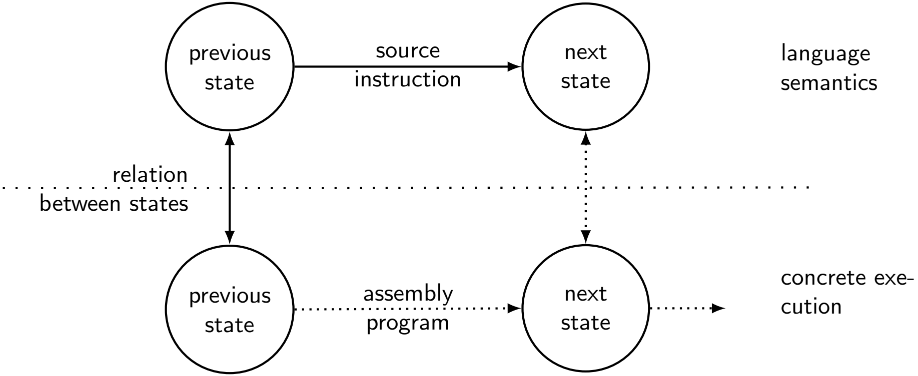
$$
\texttt{TrExp}(e_1 <<_{32} e_2) \quad = \quad \texttt{TrExp}(e_1) <<_{32} (\texttt{TrExp}(e_2)\ \&\ 31)
$$

# Translation $\mathsf{L}_2 \to$ C#minor

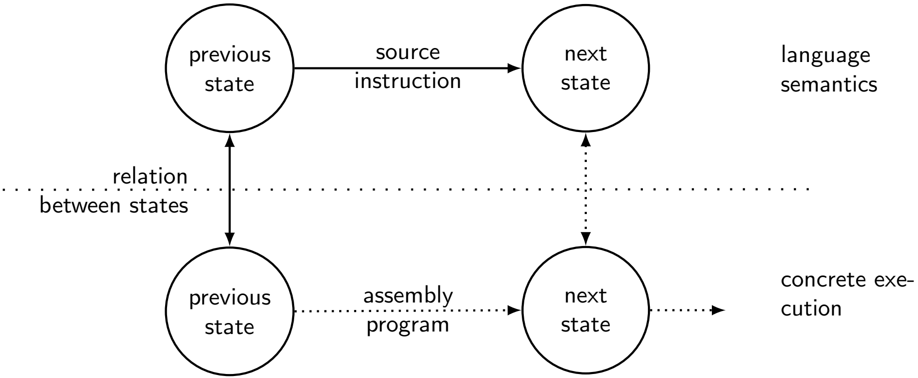
Translation from $\mathsf{L}_2$ to C#minor is mostly a 1-to-1 translation, except for the following constructions:

$$\begin{aligned}
\mathtt{TrExp}(id[e_1, \ldots, e_k]) \quad &= \quad *\big(id^t + \mathtt{sizeof}(id[0, \ldots, 0])\times \\
&\qquad ((((e_1^t \times s_2^t + e_2^t) \times s_3^t + \ldots) \ldots) \times s_k^t + e_k^t)\ \big) \\
&\quad \text{where } e^t = \mathtt{TrExp}(e) \\
\mathtt{TrExp}(e_1 <<_{32} e_2) \quad &= \quad \mathtt{TrExp}(e_1) <<_{32} (\mathtt{TrExp}(e_2)\ \&\ 31) \\
\mathtt{TrStmt}(\mathtt{error}) \quad &= \quad \mathtt{loop}\ \{\mathtt{abort}();\}
\end{aligned}$$

# Formally verified compilation



# Formally verified compilation




Every property on the source program is also verified by the generated program.

# Difficulties

- Ensure generated tests are correct and complete.
- Maintain a correspondance between our environment and the memory of C#minor.

# Translation $\mathrm{L}_2 \to$ C#minor - Proof (Visibility)

$$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$$

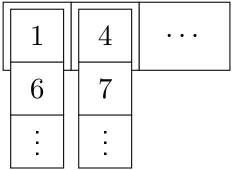
# Translation $\mathrm{L}_2 \rightarrow$ C#minor - Proof (Visibility)

$$t : \mathbb{N} \times \mathbb{P} \rightarrow \mathbb{N}^* \times \mathbb{N} \times \mathbb{V} \qquad \mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$$

# Translation $\mathrm{L}_2 \to$ C#minor - Proof (Visibility)

$$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V} \qquad \mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$$

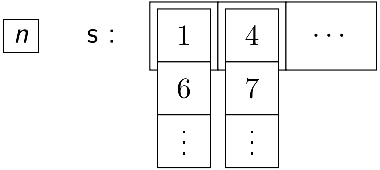
$\boxed{n}$ s :



# Translation $\mathsf{L}_2 \to$ C#minor - Proof (Visibility)

$$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$$

$$\mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$$



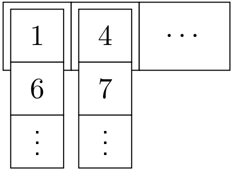

$$\begin{aligned} t(n, s) &= (42, 0, \texttt{Visible}) \\ t(n, s[0]) &= (71, 0, \texttt{Visible}) \end{aligned}$$

# Translation $L_2 \to$ C#minor - Proof (Visibility)

$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$

$\mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$

$\boxed{n}$ s :




$$t(n, s) = (42, 0, \texttt{Visible})$$
$$t(n, s[0]) = (71, 0, \texttt{Visible})$$

$\boxed{n+1}$

```
f(u: mut [[i64; n]; m], m n: u64)
```

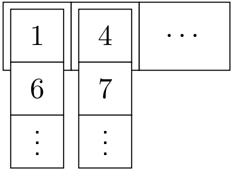
# Translation $L_2 \to$ C#minor - Proof (Visibility)

$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$

$\mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$

$\boxed{n}$

s :

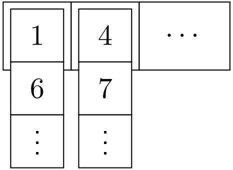



$$
\begin{aligned}
t(n, s) &= (42, 0, \texttt{Visible}) \\
t(n, s[0]) &= (71, 0, \texttt{Visible})
\end{aligned}
$$

$\boxed{n+1}$

```
f(u: mut [[i64; n]; m], m n: u64)
```

$u$ :



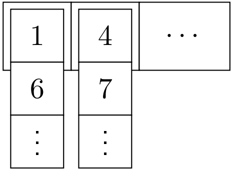
# Translation $L_2 \to$ C#minor - Proof (Visibility)

$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$

$\mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$

$\boxed{n}$

s :

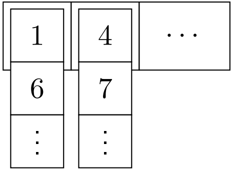



$$\begin{aligned} t(n, s) &= (42, 0, \texttt{Visible}) \\ t(n, s[0]) &= (71, 0, \texttt{Visible}) \end{aligned}$$

$\boxed{n+1}$

```
f(u: mut [[i64; n]; m], m n: u64)
```

$u$ :

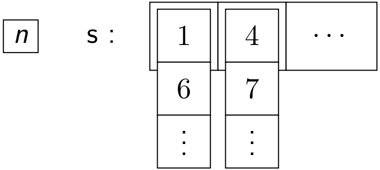



$$\begin{aligned} t'(n+1, u) &= (42, 0, \texttt{Visible}) \\ t'(n+1, u[0]) &= (71, 0, \texttt{Visible}) \end{aligned}$$

# Translation $L_2 \to$ C#minor - Proof (Visibility)

$t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$

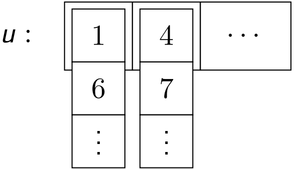
$\mathbb{V} = \{\texttt{Visible}\} \cup \{\texttt{Hidden}(p) \mid p \in \mathbb{P}\}$




$$\begin{aligned} t(n, s) &= (42, 0, \texttt{Visible}) \\ t(n, s[0]) &= (71, 0, \texttt{Visible}) \end{aligned}$$

$\boxed{n+1}$

```
f(u: mut [[i64; n]; m], m n: u64)
```




$$\begin{aligned} t'(n+1, u) &= (42, 0, \texttt{Visible}) \\ t'(n+1, u[0]) &= (71, 0, \texttt{Visible}) \\ t'(n, s) &= (42, 0, \texttt{Hidden}(u)) \\ t'(n, s[0]) &= (71, 0, \texttt{Hidden}(u[0])) \end{aligned}$$

# Translation $\mathrm{L}_2 \rightarrow$ C#minor - Proof (Invariants)

Synchronisation between environments and C#minor's memory:

$$
\begin{array}{ll}
\forall n\ p\ lv, & E_n[p] = \texttt{Varr}\ lv \rightarrow \\
& \exists b\ o\ s, \quad t(n,p) = (b,o,s) \wedge \\
& \qquad s = \texttt{Visible} \rightarrow \\
& \qquad \forall i, \quad 0 \leq i < |lv| \rightarrow \\
& \qquad\qquad M[(b, o + i \times \texttt{sizeof}(\Gamma_{F_n}[p +\!\!+ [0]]))] = \texttt{transl\_value}(lv[i])
\end{array}
$$

# Translation $\mathrm{L}_2 \rightarrow$ C#minor - Proof (Invariants)

Synchronisation between environments and C#minor's memory:

$$\begin{array}{ll}\forall n\ p\ lv, & E_n[p] = \mathtt{Varr}\ lv \rightarrow \\ & \exists b\ o\ s, \quad t(n,p) = (b,o,s) \wedge \\ & \qquad s = \mathtt{Visible} \rightarrow \\ & \qquad \forall i, \quad 0 \leq i < |lv| \rightarrow \\ & \qquad\qquad M[(b, o + i \times \mathtt{sizeof}(\Gamma_{F_n}[p +\!\!+ [0]]))] = \mathtt{transl\_value}(lv[i])\end{array}$$

Separation of visible paths in the translation function:

$$\begin{array}{ll}\forall n\ (i,l)\ b\ o, & t(n,(i,l)) = (b,o,\mathtt{Visible}) \wedge P_{F_n}(i) \geq \mathtt{Mut} \rightarrow \\ & (\forall m\ p'\ b'\ o', m < n \wedge t(m,p') = (b',o',\mathtt{Visible}) \rightarrow b \neq b') \wedge \\ & (\forall p'\ b'\ o', (i,l) \neq p' \wedge t(n,p') = (b',o',\_))) \rightarrow b \neq b')\end{array}$$

# Formal verification

Coq

```
Theorem transl_stmt_sem_preservation:
  forall p hfuncs Habort tp s s' ts t,
    ...
    transl_program' hfuncs Habort p = OK tp →
    match_states p hfuncs Habort s ts →
    step_events (genv_of_program p) s t s' →
    exists ts', plus Csharpminor.step (Genv.globalenv tp) ts t ts' ∧
                match_states p hfuncs Habort s' ts'.

Theorem transl_program_correct hfuncs (p: program):
  forall tp,
    transl_program hfuncs p = OK tp →
    forward_simulation (SemanticsBlocking.semantics p)
                       (Csharpminor.semantics tp).
```

# Stats

| Coq | Code / Spec | Proof |
|---|---|---|
| Syntax and types | 814 | 283 |
| Common semantics definitions and proofs | 1403 | 1040 |
| $L_1$ semantics | 882 | 495 |
| $L_2$ semantics | 367 | 107 |
| $L_1 \to L_2$ | 887 | 1642 |
| $L_2 \to$ C#minor | 1901 | 2910 |
| Typing | 616 | 104 |
| Safety | 1056 | 2776 |
| Miscellaneous | 979 | 936 |

# Stats

| Coq | Code / Spec | Proof |
|---|---|---|
| Syntax and types | 814 | 283 |
| Common semantics definitions and proofs | 1403 | 1040 |
| $L_1$ semantics | 882 | 495 |
| $L_2$ semantics | 367 | 107 |
| $L_1 \to L_2$ | 887 | 1642 |
| $L_2 \to$ C#minor | 1901 | 2910 |
| Typing | 616 | 104 |
| Safety | 1056 | 2776 |
| Miscellaneous | 979 | 936 |

$+ \sim 2000$ lines of OCaml (parser, type inference and simplifications)

# Generated code: addition of vectors

```
void add_vectors(a: [i64; n], b: [i64; n], dest: mut [i64; n], n: u64) {
    for i: u64 = 0 .. n {
        dest[i] = a[i] + b[i]
    }
}
```

# Generated code: addition of vectors (Assembler)

```
add_vectors: ; %rdi = a, %rsi = b, %rdx = dest, %rcx = n
   ...
  xorq  %rax, %rax    ; %rax = i ← 0
.L100:
  cmpq  %rcx, %rax    ; for loop condition
  jae .L101           ; i ⩾ n ⇒ end of loop
  cmpq  %rcx, %rax    ; array bound check
  jae .L102           ; i ⩾ n ⇒ error
  movq  0(%rdi,%rax,8), %r8  ; %r8 ← a[i]
  movq  0(%rsi,%rax,8), %r9  ; %r9 ← b[i]
  leaq  0(%r8,%r9,1), %r8    ; %r8 ← %r8 + %r9 = a[i] + b[i]
  movq  %r8, 0(%rdx,%rax,8)  ; dest[i] ← %r8 = a[i] + b[i]
  leaq  1(%rax), %rax        ; i ← i + 1
  jmp .L100
.L102:                ; translation of error
  call  abort
  jmp .L102
.L101:
   ...
  ret
```

# Conclusion

# We now have

A language

- safe
- suitable for computer algebra algorithms
- simplifying proof of programs (no aliasing, no memory, mutability)

# We now have

A language

- safe
- suitable for computer algebra algorithms
- simplifying proof of programs (no aliasing, no memory, mutability)

A formally verified compiler

- generating correct code
  semantics preservation theorem proved with Coq
- a bit of optimization

# Future work

- More constructions
  - array views
  - records
  - `malloc` / `free` (in progress)

# Future work

- More constructions
  - array views
  - records
  - `malloc` / `free` (in progress)
- Programs logic ($\Rightarrow$ functional correctness)

# Future work

- More constructions
  - array views
  - records
  - `malloc` / `free` (in progress)
- Programs logic ($\Rightarrow$ functional correctness)
- Optimizations / performance

Thanks !

# Semantics: evaluation of path

$$\text{EvPNil}\;\frac{}{E, F, \_ \vdash [] \Rightarrow []}$$

$$\text{EvScell}\;\frac{\begin{array}{c}\forall k, E, F \vdash u_k \Rightarrow \texttt{Vint}_{64}\ n_k \\ E, F \vdash \vec{i} \Rightarrow \vec{v} \\ \texttt{build\_index}\ \vec{v}\ \vec{n} = \texttt{Some}\ j \\ \texttt{valid\_index}\ \vec{v}\ \vec{n}\end{array}}{E, F, \vec{u} :: \vec{l} \vdash (\texttt{Scell}\ \vec{i}) :: \vec{s} \Rightarrow (\texttt{Pcell}\ j) :: \vec{z}}$$

# Semantics: evaluation of path - error cases

$$\text{ErrP1}\ \frac{\begin{array}{c}\forall k, E, F \vdash u_k \Rightarrow \texttt{Vint}_{64}\ n_k \\ E, F \vdash \vec{i} \Rightarrow \vec{v} \\ \texttt{build\_index}\ \ \vec{v}\ \ \vec{n} = \texttt{Some}\ \ j \\ \neg\texttt{valid\_index}\ \ \vec{v}\ \ \vec{n}\end{array}}{E, F, \vec{u} :: \vec{l} \vdash (\texttt{Scell}\ \ \vec{i}) :: \vec{s} \Rightarrow \texttt{error}}$$

$$\text{ErrP2}\ \frac{\begin{array}{c}\forall k, E, F \vdash u_k \Rightarrow \texttt{Vint}_{64}\ n_k \\ E, F \vdash \vec{i} \Rightarrow \texttt{error}\end{array}}{E, F, \vec{u} :: \vec{l} \vdash (\texttt{Scell}\ \ \vec{i}) :: \vec{s} \Rightarrow \texttt{error}}$$

# Example: Eratosthene's sieve

```
fun eratosthene(prime: mut [bool; N], N: u64) {
  if N < 2 return;
  prime[0u32] = false;
  prime[1u32] = false;
  for k: u64 = 4 .. N step 2
    prime[k] = false;
  let i: u64 = 3;
  while (i * i < N) {
    if prime[i] {
      for j: u64 = i .. (N / i + 1) step 2
        prime[i * j] = false;
    }
    i = i + 1
  }
}
```

# Example: Block Matrix Multiplication

```
fun block_mul_matrix(a: [i64; m, n], b: [i64; n, p], dest: mut [i64; m, p],
                     m n p bs: u64) {
  let s: i64 = 0;
  for I: u64 = 0 .. m step bs {
    let Imax: u64 = min(m, I + bs);
    for J: u64 = 0 .. p step bs {
      let Jmax: u64 = min(p, J + bs);
      for K: u64 = 0 .. n step bs {
        let Kmax: u64 = min(n, K + bs);
        for i: u64 = I .. Imax
          for j: u64 = J .. Jmax {
            s = 0;
            for k: u64 = K .. Kmax
              s = s + a[i, k] * b[k, j];
            dest[i, j] = dest[i, j] + s
          }
      }
    }
  }
}
```

# Example: Gauss

```
fun gauss(A: mut [f64; n, m], n m: u64) {
  let r: u64 = 0;
  for j: i64 = 0 .. m {
    let k: i64 = search_max_abs(A, n, m, r, j);
    if (k = -1) break;
    if A[k, j] ≠ 0. {
      divide_line_by_const(A, n, m, (u64) k, A[k, j]);
      if (u64) k ≠ r
        exchange_lines(A, n, m, (u64) k, r);
      for i: u64 = 0 .. n {
        if i ≠ r
          add_lines(A, n, m, i, j, - A[i, j]);
      }
      r = r + 1
    }
  }
}
```

# Example: Dot product of complex vectors (BLAS)

```
fun zdotu(n: u64, zx: [f64; 2*n], incx: i32,
                  zy: [f64; 2*n], incy: i32, res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;
  if incx == 1 && incy == 1 {
    for i: u64 = 0 .. (2 * n) step 2 {
      res[0] = res[0] + zx[i] * zy[i] - zx[i+1] * zy[i+1];
      res[1] = res[1] + zx[i+1] * zy[i] + zx[i] * zy[i+1];
    }
  } else {
    let ix: i32 = 1; let iy: i32 = 1;
    if (incx < 0) ix = (-(i32)(2*n)+1)*incx + 1;
    if (incy < 0) iy = (-(i32)(2*n)+1)*incy + 1;
    for i: u64 = 0 .. n {
      res[0] = res[0] + zx[ix] * zy[iy] - zx[ix+1] * zy[iy+1];
      res[1] = res[1] + zx[ix+1] * zy[iy] + zx[ix] * zy[iy+1];
      ix = ix + 2 * incx; iy = iy + 2 * incy;
    } } }
```

# Translation $\mathsf{L}_1 \to \mathsf{L}_2$: test generation

$$\mathtt{ET}\big(\mathtt{divu}(e_1, e_2)\big) \quad = \quad \mathtt{ET}(e_1) \mathbin{+\!\!+} \mathtt{ET}(e_2) \mathbin{+\!\!+} (e_2 \neq 0)$$

The order of tests is important.

# Translation $\mathsf{L}_1 \to \mathsf{L}_2$: test generation

$$
\begin{aligned}
\mathtt{ET}\big(\mathtt{divu}(e_1, e_2)\big) &= \mathtt{ET}(e_1) +\!\!+ \mathtt{ET}(e_2) +\!\!+ (e_2 \neq 0) \\
\mathtt{ET}\big(\mathit{divs}(e_1, e_2)\big) &= \mathtt{ET}(e_1) +\!\!+ \mathtt{ET}(e_2) +\!\!+ \\
&\quad (e_2 \neq 0 \wedge (e_1 \neq \mathtt{min\_sint} \vee e_2 \neq -1))
\end{aligned}
$$

The order of tests is important.

# Translation $\mathsf{L}_1 \to \mathsf{L}_2$: test generation

$$
\begin{array}{lcl}
\mathtt{ET}(\mathtt{divu}(e_1, e_2)) & = & \mathtt{ET}(e_1) +\!\!+ \mathtt{ET}(e_2) +\!\!+ (e_2 \neq 0) \\
\mathtt{ET}(\mathit{divs}(e_1, e_2)) & = & \mathtt{ET}(e_1) +\!\!+ \mathtt{ET}(e_2) +\!\!+ \\
 & & (e_2 \neq 0 \wedge (e_1 \neq \mathtt{min\_sint} \vee e_2 \neq -1)) \\
\mathtt{ET}((\mathtt{int}_{32,\mathtt{Unsigned}} \to \mathtt{int}_{64,\mathtt{Unsigned}})e) & = & \mathtt{ET}(e) \\
\mathtt{ET}((\mathtt{int}_{64,\mathtt{Unsigned}} \to \mathtt{int}_{32,\mathtt{Unsigned}})e) & = & \mathtt{ET}(e) \\
\mathtt{ET}((\mathtt{int} \to \mathtt{float})e) & = & \mathtt{ET}(e) \\
\mathtt{ET}((\mathtt{float}_{32} \to \mathtt{int}_{32,\mathtt{Signed}})e) & = & \mathtt{ET}(e) +\!\!+ (-2^{31} \leq e < 2^{31}) \\
\mathtt{ET}((\mathtt{float}_{64} \to \mathtt{int}_{32,\mathtt{Signed}})e) & = & \mathtt{ET}(e) +\!\!+ (-2^{31} - 1 < e < 2^{31}) \\
\mathtt{ET}((\mathtt{float}_{32} \to \mathtt{int}_{32,\mathtt{Unsigned}})e) & = & \mathtt{ET}(e) +\!\!+ (-1 < e < 2^{32}) \\
\mathtt{ET}((\mathtt{float}_{64} \to \mathtt{int}_{32,\mathtt{Unsigned}})e) & = & \mathtt{ET}(e) +\!\!+ (-1 < e < 2^{32}) \\
\mathtt{ET}(x[i_1, \ldots, i_k]) & = & \mathtt{ET}(i_1) +\!\!+ \ldots +\!\!+ \mathtt{ET}(i_k) +\!\!+ \\
 & & (i_1 <_u s_1) +\!\!+ \ldots +\!\!+ (i_k <_u s_k) \\
 & & \text{where } s_1, \ldots, s_k \text{ are the size variables of } x
\end{array}
$$

The order of tests is important.

# Generated code: addition of vectors ($L_1$)

```
void add_vectors(a: [i64; n], b: [i64; n], dest: mut [i64; n], n: u64)
{
  u64 $8; u64 $9; u64 i;
  /* $9 = n; */
  i = 0u64;
  $8 = n;
  while true {
    if (i <u $8) {
      dest[i] = a[i] + b[i];
      i = i + 1u64;
    } else break;
  }
}
```

# Generated code: addition of vectors ($\mathsf{L}_2$)

```
void add_vectors(a: [i64; n], b: [i64; n], dest: mut [i64; n], n: u64)
{
  u64 $8; u64 $9; u64 i;
  /* $9 = n; */
  i = 0u64;
  $8 = n;
  while true {
    if (i <u $8) {
      assert (i <u $9);
      dest[i] = a[i] + b[i];
      i = i + 1u64;
    } else break;
  }
}
```

# Generated code: addition of vectors (Cminor)

```
"add_vectors"('a', 'b', 'dest', 'n') : long → long → long → long → void
{
  var '$9', '$8', 'i';
  goto 'code';
  'error': "abort"() : void;
           goto 'error';
  'code': '$9' = 'n'; '$9' = 'n'; '$9' = 'n'; 'i' = 0LL; '$8' = 'n';
          {{ loop {
              {{ if ('i' <lu '$8') {
                   if ('i' <lu '$9') { /*skip*/ }
                   else { goto 'error'; }
                   int64['dest' +l 8LL *l 'i'] =
                     int64['a' +l 8LL *l 'i'] +l int64['b' +l 8LL *l 'i'];
                   'i' = 'i' +l 1LL;
                 } else { exit 1; }
              }}
            }
          }}
}
```