A Safe Low-level Language for Computer Algebra and its Formally Verified Compiler

Josué Moreau, Guillaume Melquiond

A Safe Low-level Language for Computer Algebra and its Formally Verified Compiler

JOSUÉ MOREAU, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, France
GUILLAUME MELQUIOND, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, France

This article describes a programming language for writing low-level libraries for computer algebra systems. Such libraries (GMP, BLAS/LAPACK, etc) are usually written in C, Fortran, and Assembly, and make heavy use of arrays and pointers. The proposed language, halfway between C and Rust, is designed to be safe and to ease the deductive verification of programs, while being low-level enough to be suitable for this kind of computationally intensive applications. This article also describes a compiler for this language, based on CompCert. The safety of the language has been formally proved using the Coq proof assistant, and so has the property of semantics preservation for the compiler. While the language is not yet feature-complete, this article shows what it entails to design a new domain-specific programming language along its formally verified compiler.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Logic and verification**; *Semantics and reasoning*; • **Mathematics of computing** → Mathematical software.

Additional Key Words and Phrases: programming language, compiler, formal proof, safety

1 INTRODUCTION

While computer algebra systems are often used to perform symbolic computations, they internally rely on various low-level libraries to manipulate data structures: GMP (arithmetic over long integers), BLAS/LAPACK (linear algebra), FFTW (fast Fourier transform), and so on. These state-of-the-art libraries use the C programming language for GMP and FFTW, and Fortran for BLAS and LAPACK, not counting some large pieces of architecture-specific Assembly code for GMP and BLAS. Inertia is the main reason for not switching these libraries to more modern languages, but another reason is that these languages do not get in the way of the programmer (possibly by disregarding safety). As an example, let us consider the following low-level function from GMP:

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

This function computes the square of an n-word integer stored in the memory range $[\mathtt{s1p}, \mathtt{s1p}+n)$ and stores the result in the memory range $[\mathtt{rp}, \mathtt{rp}+2n)$. If the function was written in a modern language like Rust, either its signature would have to be changed to use *slices*, *i.e.*, pairs of pointer and size (thus breaking backward compatibility), or it would have to use the *unsafe* fragment of the language. One of the reasons for having to use the unsafe fragment is that the above signature lacks a lot of information regarding the validity range of the pointer arguments. So, the compiler has no way to check that the caller of the function did pass suitable pointers as arguments. With the introduction of the C99 revision of the language, the signature could have been made more explicit (at the expense of compatibility, since the order of arguments changes):

```
void mpn_sqr(mp_size_t n, mp_limb_t rp[static restrict 2*n],
             const mp_limb_t s1p[static n]);
```

The `static` keyword instructs the compiler that the pointers are valid for the given range, while `restrict` instructs that no access through the pointer `s1p` overlaps accesses through the restricted pointer `rp`. The latter paves the way for optimizations, since the compiler can prefetch the reads

Authors' addresses: Josué Moreau, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France, josue.moreau@inria.fr; Guillaume Melquiond, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France, guillaume.melquiond@inria.fr.

through `s1p` (*e.g.*, for vectorization purpose) without worrying about a potential data dependency with the writes performed through `rp`. Indeed, the `const` keyword by itself does not guarantee that the data pointed by `s1p` is immutable, as the C language allows the zones pointed by `rp` and `s1p` to overlap, hence the need for the `restrict` keyword. As a matter of fact, GMP's user documentation has to explicitly state that "no overlap is permitted between the destination and the source."

While the C99 language makes it possible to precisely state the expectations of the `mpn_sqr` function by adding new undefined behaviors, the improved signature does not bring much to the calling function. Indeed, despite the `const` keyword, `mpn_sqr` could well mutate the data pointed by `s1p` without triggering any undefined behavior. It could also mutate data outside the range specified for `rp`. Even if we disregard this lack of precision of the signature from the point of view of the caller, the whole idea of adding even more undefined behaviors to the language might not seem like a progress, as compilers will often be unable to detect any misuse of the function.

Since the algorithms that appear in the low-level libraries used by computer algebra systems often use a very specific subset of the programming language, this motivates the design of a language dedicated to them. It should fulfill the following requirements:

- It should be close enough from low-level languages like C and Fortran, so as to not add a barrier on entry. Array types should have explicit sizes, so as to accommodate the signatures of functions found in these libraries. Moreover, multi-dimensional arrays should be supported, as they are ubiquitous in BLAS-like libraries.
- The language should be safe, that is, a well-typed program should have no undefined behavior. That does not mean that, for example, the type system prevents out-of-bounds array accesses, but rather that they cannot cause a silent memory corruption. Moreover, this safety should not come at the expense of the type system, which should stay simple enough (*e.g.*, no Rust-like lifetime).
- The language should be deterministic, for reproducibility and portability purposes. Even stronger, its semantics should make it straightforward to perform some deductive program verification of the implemented algorithms, without having to rely on a dedicated verification logic, *e.g.*, separation logic, if possible. In particular, no undue mutation should ever happen because of pointer aliasing.

This article presents a tentative design for such a programming language, halfway between C and Rust, whose main specificity is that its copy-restore semantics is free of any kind of memory model and all the related troubles. But more importantly, this new language comes with a formal semantics, a formal proof of type soundness that is mechanically checked with the Coq proof assistant, and a formally verified compiler built on top of the CompCert compiler [Leroy 2009b]. Having a formally verified compiler means that, not only there is a working compiler for the language, but there is also a Coq proof that the semantics of the original source code is preserved by the compiler transformations until Assembly code. In particular, this formally guarantees that there is no miscompilation.

The language is still in its early stage and various features are missing. Most notably, functions cannot partition mutable arrays to implement divide-and-conquer algorithms (*e.g.*, fast integer multiplication) in a way that would make their formal verification easier. Nonetheless, this work should still be representative of the benefits and costs of formalizing a new language in parallel to the development of a formally verified compiler, rather than performing a formal verification after the fact.

Section 2 gives an overview of the syntax of the language and a more detailed presentation of its features, as well as its informal relation to the C language. Section 3 then presents the architecture of our compiler and how the input language is being compiled to one of CompCert's intermediate

languages. In particular, this involves translating all the arguments passed by copy-restore into plain pointers. Section 4 gives a more formal view of the semantics of the input language as well as its type system. Section 5 explains how the soundness of the type system and the correctness of the compilation passes were formally verified using the Coq proof assistant. Section 6 explains how this article compares to some related works. Finally, Section 7 describes some missing features of the language and what their expected cost will be, verification-wise.

2 LANGUAGE

To stay close to the C and Fortran languages, our language is also an imperative one. Therefore, it implements the traditional control structures, that is, `if` conditionals, `for` loops (increasing or decreasing, with a stepping possibly larger than 1), `while` loops, assignments, function calls, and `return`. Currently, the primitive types are booleans, integers (8, 16, 32, and 64 bits), and floating-point numbers (32 and 64 bits). All the functions are mutually recursive, so there is no need to declare a function, except for external functions.

The language does not have any undefined behavior. Exceptional behaviors such as out-of-bound accesses or division by zero lead to a runtime error, which aborts the execution. All the arithmetic operators are modular, and shifts apply a modulo 32 or 64 on their right-hand operand.

To conclude this quick presentation of our language, here is the traditional, imperative implementation of the factorial function:

```
fun fact(n: i32) -> i32 {
  let x: i32 = 1;
  while n > 0 { x = x * n; n = n - 1; }
  return x;
}
```

As they are a critical piece of the libraries we are targeting, Section 2.1 focuses on the topic of arrays. Section 2.2 then explains how this language can be compiled and what the advantages are. Finally, Section 2.3 considers a realistic example taken from the BLAS library.

2.1 Arrays

First and foremost, most computer algebra libraries are built on arrays and, more generally, multidimensional arrays. They are different from arrays of arrays, as the latter are represented in memory as arrays of pointers to arrays, while multidimensional arrays are represented as a one-dimensional array. The representation is row-major, that is, `t[x, y]` is interpreted as `t[x * m + y]`, where `t` is a $n \times m$ array.

In high-level libraries, multidimensional arrays are often represented using a structure that contains both the elements and the shape. In the case of low-level libraries, such as BLAS or the *mpn* layer of GMP, array sizes are passed explicitly as function arguments. This permits more flexibility, since one can pass a smaller size than the allocated space. In a similar way, function signatures in our language use explicit size expressions. For example, matrix multiplication could be declared as follows:

```
fun matrix_mul(a:[i32;m,n], b:[i32;n,p], dest: mut[i32;m,p], m n p:u64)
```

As illustrated by this example, having explicit size expressions makes it possible to relate the array sizes of several array declarations, and thus to stay close to the mathematical definition of linear algebra operations. There is another advantage, illustrated by the simpler example of vector addition below. Since all three arrays `a`, `b`, and `dest` have the same size n, the validity of the corresponding accesses do not have to be checked separately.

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

To ensure the safety of the language, when an array is accessed outside of its bounds, we want the execution of the program to be interrupted, contrarily to C where it is just an undefined behavior and where it might thus silently corrupt memory. For a similar reason, passing an array of an unexpected size as an argument also interrupts the program, as it could make an array look larger than it actually is. Array operations are not the only ones that might interrupt the execution. For example, division by zero and casting a floating-point number f to an integer when the result would end up outside the range of the integer type will also do so. The language also provides an `error` instruction which always interrupts the execution.

Finally, in order to help reasoning about programs, arrays are always passed by copy-restore. When entering a function, the code behaves as if a deep copy of the array arguments was performed, and, when returning from the function, the arrays that were marked with the `mut` keyword are copied back to the caller. This gives a clearer semantics on the effects of function calls. Indeed, after a function call, only `mut` arrays are restored, so we can clearly state that other arrays are not modified, whether passed or not. When performing a deductive verification, post-conditions of the form $\forall i,\ 0 \le i < \texttt{length}(t) \Rightarrow t[i] = (\texttt{old t})[i]$ (here, `old t` refers to the value of `t` before the function call) are no longer necessary.

Still, there is a problem at function exit if there was some aliasing between a mutable array and some other array also passed as argument. It is not clear how the mutable array should be restored, if at all. To avoid this issue, creating an alias between a mutable argument and any other argument is a compile-time error.

2.2 Relation to the C language

The language, as described above, would have poor performance if the compiled program was actually performing a deep copy of array arguments at each function call. In order to address this issue, we take advantage from the fact we previously stated: at function call, a mutable argument is not aliased with any other passed array. This entails that, at any time, there is only one way to access a mutable array. As a consequence, there is no observable difference between directly modifying the original array and modifying a copy of the array which will overwrite the original array at function exit. To sum up, our language semantics states that arguments are passed by copy-restore, as it is easier to reason about, but the compiled program will pass them by reference, to avoid the costly deep copies. The correctness of this transformation has been formally proved, as explained in Section 5.

As a consequence of using pass-by-reference, our language is very close to the C language, while it adds some key features that guarantee that any well-typed program is safe. Another advantage of being close to C is that we can use the same calling convention as in C. As a consequence, one can easily use some existing C libraries directly in programs written in our language. Alias and mutability information can be specified for external functions, and they will be checked before calling. This ensures safe calls to external libraries. Conversely, functions written in our language can be directly called from C code, which is the primary use for our language. For instance, the above function `add_vectors` has the following C signature:

```
void add_vectors(const int64_t* a, const int64_t* b,
                 int64_t* restrict dest, uint64_t n);
```

```
fun zdotu(n: i32, zx: [f64; (u64) (1 + (n - 1) * incx), 2], incx: i32,
                  zy: [f64; (u64) (1 + (n - 1) * incy), 2], incy: i32,
          res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;

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

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

Fig. 1. Implementation of the zdotu function of BLAS

2.3 A representative example

Let us look at a larger example, coming from the BLAS library. The `zdotu` function is responsible for computing the scalar product of two vectors with complex components. Figure 1 shows the translation of this function to our language. As can be seen by looking at the original Fortran code,[1] the translation of the body is mostly mechanical and it features all the implementation details, including the arguments `incx` and `incy` which describe the storage spacing between the components of the input vectors. The most notable difference in the code is the way the indices `ix` and `iy` are computed, since arrays start at index 0 in our language instead of 1 for Fortran.

Since the compiler does not yet provide a primitive type for complex numbers, we represent them as arrays of size 2. As a consequence, the input vectors `zx` and `zy` are multidimensional arrays. Nevertheless, they have the exact same memory representation as the vectors from the original implementation. Another consequence of not having a primitive type for complex numbers is that, since functions cannot yet return arrays, our implementation of `zdotu` cannot directly return the scalar product. Instead, we have to tweak the signature a bit so that the returned complex number is passed as a mutable argument `res`.

[1] https://www.netlib.org/lapack/explore-html/d1/dcc/group__dot_ga6b0b69474b384d45fc4c7b1f7ec5959f.html

An interesting point of `zdotu` is that, while the Fortran code declares the input arguments `zx` and `zy` as being complex arrays of unspecified size, its documentation explicitly states that `zx` has size $1 + (n - 1) \cdot |\texttt{incx}|$, and similarly for `zy`. Our language does not yet support calling functions in size expressions, so we can not use an absolute value there. But other than that, `zx` and `zy` are appropriately declared as having the originally documented size.

In our language, those size expressions are not just documentation; they are part of the semantics. In particular, a caller of `zdotu` has to dynamically check that the size of the array arguments `zx` and `zy` are compatible with the values of the other arguments: `n`, `incx`, and `incy`. A failed check is not an undefined behavior but a runtime error. Obviously, this constraint only applies for caller functions written in our language. If `zdotu` is directly called as an external function from C for instance, then the body of `zdotu` can only assume that the arrays have the proper size.

Finally, due to the copy-restore semantics, the input arrays `zx`, `zy`, and `res` behave as if they were physically distinct. In particular, it means that writes to `res` modify neither `zx` nor `zy`. As a consequence, an optimizing compiler could eagerly perform the reads from `zx` and `zy`, thus offering opportunities for vectorization (see Section 3.2).

3 COMPILER ARCHITECTURE

We now present the architecture of our compiler, which is divided into 3 parts. The first one, written in OCaml, is made of the parser, some type inference and checking, and a few transformations to simplify the source code: lazy operators are removed, function arguments are turned into l-values, and the program is put into A-normal form. The resulting intermediate language, which we call L_1, is summarized in Figure 2. The most notable difference with the surface language L_0 is that the types of arrays no longer mention their sizes, so they are just like pointers. This first part of the compiler is quite straightforward, so we will not detail it any further.

The second part of our compiler, detailed in Section 3.1, translates the intermediate language to C#minor,[2] one of the high-level intermediate language of the CompCert C compiler [Leroy 2009a,b]. The third part of the compiler, from C#minor to Assembly, relies on the CompCert compiler in a straightforward way. These last two parts correspond to the formally verified part of our work, whose proofs will be presented in Section 5.2.

The resulting compiler supports the same 64-bit architectures as CompCert, that is, x86-64 and Aarch64. As CompCert performs few optimizations, our compiler can also output C code, which can be sent to an optimizing compiler. This is described in Section 3.2.

3.1 Sound translation to C#minor

The second part of the compiler is written in the Gallina language of the Coq proof assistant. This choice comes out of the fact that CompCert and its formalization are written using Coq, which we will use later to formally prove the correctness of our compiler (see Section 5). There is a second type checker, written in Gallina, which checks the L_1 program before translating it to C#minor. The typing rules are the same as for the type checker written in OCaml, but they are reformulated in a way that is more suitable for the safety proof. Also, the error messages of the second checker are much less precise than the OCaml one.

The C#minor language has a semantics whose memory model uses pointers [Leroy and Blazy 2008]. The translation from L_1 to C#minor achieves the goal announced in Section 2.2, that is, the translation of function arguments passed by copy-restore into arguments passed by reference. As C#minor's semantics has undefined behaviors, we chose to split the translation into two passes. The

[2] C#minor's semantics is formally described in the CompCert commented Coq development : https://compcert.org/doc/html/compcert.cfrontend.Csharpminor.html

$$
\begin{array}{lcl}
\tau & ::= & \texttt{void} \\
& | & \texttt{bool} \\
& | & \texttt{int}_{sz,sg} \\
& | & \texttt{float}_{fsz} \\
& | & \tau\ \texttt{arr} \\
sz & ::= & 8 \mid 16 \mid 32 \mid 64 \\
sg & ::= & \texttt{Signed} \mid \texttt{Unsigned} \\
fsz & ::= & 32 \mid 64
\end{array}
\qquad
\begin{array}{lcll}
e & ::= & c & \text{constants} \\
& | & (\tau_1 \to \tau_2)e & \text{cast from type } \tau_1 \text{ to } \tau_2 \\
& | & \diamond e & \text{unary operations (not, neg)} \\
& | & e_1 \diamond e_2 & \text{binary operations } (+, -, \times, /, \gg, \ldots) \\
& | & p & \text{read from local environment} \\
p & ::= & id \cdot \vec{q} & \text{path into local environment} \\
q & ::= & [\vec{e}] & \text{multi-dimensional addressing}
\end{array}
$$

$$
\begin{array}{lcll}
s & ::= & \texttt{skip} \mid s_1; s_2 & \text{sequence} \\
& | & p \leftarrow e & \text{assignment} \\
& | & id^? \leftarrow f(p_1, \ldots, p_n) & \text{function call} \\
& | & id \leftarrow \texttt{alloc} \mid \texttt{free}\ id & \text{dynamic allocation} \\
& | & \texttt{return}\ e^? & \text{early return} \\
& | & \texttt{assert}\ e \mid \texttt{error} & \text{interruption} \\
& | & \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\} & \text{conditional} \\
& | & \texttt{loop}\ \{s\} \mid \texttt{break} \mid \texttt{continue} & \text{loop}
\end{array}
$$

Fig. 2. Types, expressions, and statements of the intermediate languages L_1 and L_2

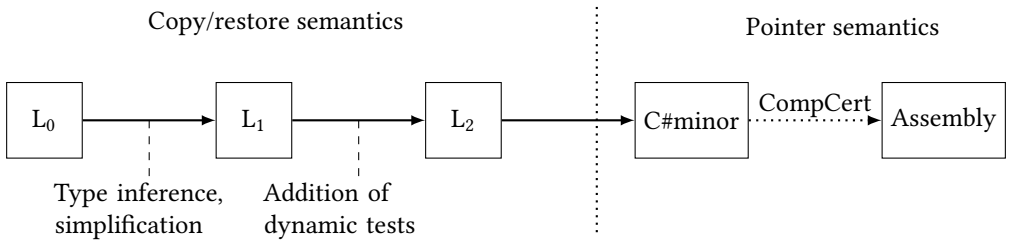

Fig. 3. Compiler architecture

first one translates L_1 to a language, say L_2, which has the same constructions as L_1 (see Figure 2), but their semantics is no longer safe. For instance, accessing an array outside of its bounds, dividing by zero, or performing an invalid cast are now undefined behaviors. The second pass goes from L_2 to C#minor. Our compiler architecture is summarized in Figure 3.

Because the constructions of L_2 are unsafe, directly translating an array access from L_1 to L_2 is incorrect. Indeed, in case of an out-of-bound access, the L_1 program would have a defined behavior (an error), while the L_2 program would have no defined behavior. To ensure that the semantics of the program is preserved, we chose to add dynamic assertions to the L_2 program. They check the validity conditions of expressions that have undefined behaviors and, in case they are not verified, interrupts the program (as would the `error` instruction).

An overview of the first pass, from L_1 to L_2, is presented in Figure 4. The main function is `TR`, which translates instructions by guarding them with the necessary and sufficient assertions to ensure that these instructions do not raise an error. The tests corresponding to these assertions is generated by the functions `ET` and `CT`. Function `ET` takes an expression and returns the corresponding list of tests. Function `CT` generates the tests that ensure that array sizes are preserved across function calls. Since its definition is long and hardly interesting, we do not show it here.[3] Finally, the `guarded`

[3]It can be found in the files `NBtoB.v` and `Validity.v` of the compiler.

$$
\begin{array}{ll}
\mathtt{ET}(c) & = [] \\
\mathtt{ET}((\mathtt{i32} \to \mathtt{f32})e) & = \mathtt{ET}(e) \\
\mathtt{ET}((\mathtt{f32} \to \mathtt{i32})e) & = \mathtt{ET}(e) +\!\!+ [-2^{31} \le e < 2^{31}] \\
\mathtt{ET}((\mathtt{f64} \to \mathtt{i32})e) & = \mathtt{ET}(e) +\!\!+ [-2^{31} - 1 < e < 2^{31}] \\
\mathtt{ET}((\mathtt{f64} \to \mathtt{i32})e) & = \mathtt{ET}(e) +\!\!+ [-1 \le e < 2^{32}] \\
\mathtt{ET}(\mathrm{divs}(e_1, e_2)) & = \mathtt{ET}(e_1) +\!\!+ \mathtt{ET}(e_2) +\!\!+ [e_2 \neq 0, e_1 \neq \mathtt{min\_signed} \vee e_2 \neq -1] \\
\mathtt{ET}(\mathtt{t}[\vec{e}]) & = \mathtt{ET}(e_1) +\!\!+ \ldots +\!\!+ \mathtt{ET}(e_k) +\!\!+ [e_1 <_u s_1, \ldots, e_k <_u s_k] \\
 & \quad \text{where } s_1 \times \ldots \times s_k \text{ is the size of the multidimensional array } \mathtt{t}.
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{TR}(\mathtt{x} \leftarrow e) & = \mathtt{guarded}(\mathtt{ET}(e), \mathtt{x} \leftarrow e) \\
\mathtt{TR}(\mathtt{x} \leftarrow f(\vec{a})) & = \mathtt{guarded}(\mathtt{ET}(\vec{a}) +\!\!+ \mathtt{CT}(f, \vec{a}), \mathtt{x} \leftarrow f(\vec{a})) \\
\mathtt{TR}(\mathtt{if}\ c\ \{s_1\}\ \mathtt{else}\ \{s_2\}) & = \mathtt{guarded}(\mathtt{ET}(c), \mathtt{if}\ c\ \{\mathtt{TR}(s_1)\}\ \mathtt{else}\ \{\mathtt{TR}(s_2)\}) \\
\mathtt{TR}(s_1; s_2) & = \mathtt{TR}(s_1); \mathtt{TR}(s_2) \\
\mathtt{TR}(\mathtt{return}\ e) & = \mathtt{guarded}(\mathtt{ET}(e), \mathtt{return}\ e)
\end{array}
$$

Fig. 4. Translation from L_1 to L_2

function takes a list of tests and an instruction; it removes all the tests that are redundant; and it prepends the given instruction with the assertions corresponding to all the remaining tests. For instance, the statement `dest[i] = a[i] + b[i]` from the `add_vectors` function of Section 2.1 should be guarded by three out-of-bounds checks, but the `guarded` function notices that two of them are superfluous. So, `TR` ends up translating the loop body into the following two statements:

```
assert (i < n);          // unsigned comparison
dest[i] = a[i] + b[i];
```

The second pass, from L_2 to C#minor, is mostly a 1-to-1 translation, except that it turns any array access $\mathtt{t}[i_1, ..., i_n]$ into some arithmetic on pointers using the row-major interpretation:

$$*(t + ((i_1 \cdot s_2 + i_2) \cdot s_3 + \ldots))$$

where $s_1 \times \ldots \times s_n$ is the size of the multidimensional array `t`. Another change introduced from L_2 to C#minor is the translation of the `error` instruction into `loop { abort() }`, since CompCert has no dedicated instruction. The reason for this seemingly pointless loop is that CompCert's semantics assumes that external functions might return.

3.2 Untrusted C backend

Having a fully verified compilation chain from L_1 to Assembly guarantees that our language is not just an abstract object but can actually be compiled. This also guarantees that the generated code was not miscompiled and behave in the way the formal semantics dictates. In most situations however, users are more concerned about the performances of the generated code than having a formal proof that it was correctly compiled.

Therefore, we have also implemented a way for our compiler to output actual C code, which can then be compiled by a separate compiler. More precisely, instead of generating C#minor, C code is produced from the L_2 code, which is again mostly a 1-to-1 translation. That does not make it completely trivial though. Indeed, we have to make sure that the proper arithmetic operations are used. For instance, in L_2 and in C#minor, arithmetic operations on signed integers are fully defined, but this is not the case for C, so we cannot blindly translate them. It would therefore be interesting to formally verify that this translation from L_2 to C is correct.

Table 1. Relative timings of zdotu compared to BLAS. Smaller is better

	unmodified	without `restrict`	simple arrays	extra assertions
Fully verified chain	2.23	–	2.28	2.21
C output + GCC	1.04	1.30	1.29	1.04
C output + LLVM	0.92	1.31	1.20	0.81

The formal proof that the translation from L_2 to C#minor preserves semantics guarantees that the arrays passed as mutable inputs to a function are disjoint from any other (see Section 5.2). This means that we can tag the corresponding pointers with the `restrict` keyword when translating to C. In particular, on the `zdotu` example of Section 2.3, both GCC and LLVM are able to take advantage of it to vectorize the body of the loops. We now focus on the case where `incx` = `incy` = 1, as it is the most interesting loop, optimization-wise.

First, it should be noted that the generated C code for `zdotu` contains numerous assertions, which checks that none of the array accesses are out of bounds, as warranted by the semantics of L_1. Fortunately, both compilers are able to eliminate most of them. Two of these assertions, however, remain in the body of the loop. Indeed, it seems that neither compiler is able to deduce from `incx` = 1 that the input array `zx` has actually size `n` (and similarly for `zy`) and thus that all the accesses are in bounds. Fortunately, having two fully predicted branches per loop iteration does not cause too much of a slowdown, so our implementation of `zdotu` is competitive with the Fortran code from BLAS.

We have also tested an implementation based on dimension-1 arrays of size $2n$ in place of multidimensional arrays of size $n \times 2$. Not only does it make the source code less readable, but it also seems to confuse the compilers, which are no longer able to eliminate most of the assertions, thus negatively impacting performances. While our implementation based on multidimensional arrays is competitive with the original one, despite the remaining assertions, we can do a bit better. Indeed, we can add to our code the following two assertions before the loop:

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

Obviously, this adds two more assertions to the generated code, but now LLVM is able to completely optimize the assertions from the body of the loop, thus making the resulting code faster.

Table 1 compares the performance of `zdotu` when compiled with CompCert, GCC 13.2.1, and LLVM 16.0.6 as backends. In the case of CompCert, this is our fully verified chain from L_1 to Assembly presented above, while for GCC and LLVM, the code goes through the unverified C backend. The original code from BLAS (LAPACK 3.12.0) is taken as the baseline; values larger than 1 mean slower than BLAS and smaller than 1 mean faster. Several variants of the code are exercised on vectors of size 1000 with consecutive components (*i.e.*, `incx` = `incy` = 1). The "unmodified" column shows the measured time for the code of Section 2.3. The following columns respectively shows the timing if `restrict` is not emitted, if dimension-1 arrays are used, and if extra assertions are put before the loop. For GCC and LLVM, the optimization level is `-O2` and no other flag is passed.

4 FORMAL SEMANTICS

Let us now give a formal definition of the languages L_1 and L_2. Section 4.1 focuses on the semantics of L_1. The semantics of L_2 is identical, except for rules that lead to an error due to a broken precondition, *e.g.*, out-of-bound accesses. These L_1 rules, recognizable by their suffix Err, are not part of L_2's semantics, so the corresponding behaviors are undefined; the only error-raising statements of L_2 are `error` and `assert`. We also present, in Section 4.2, the type system of L_1, which guarantees the absence of undefined behaviors in programs.

4.1 Semantics

As explained in Section 2.1, our language passes arguments by copy-restore. And since we forbid to take a pointer to a value, the semantics of our language does not need any memory model. It only manipulates a local environment containing arbitrarily deep, structured values. In other words, the value of an array is not comprised of an identifier that designates a memory block containing the actual content of the array, as it is the case in CompCert's memory model. In our semantics, the value of an array is directly the list of the values of its elements. For instance, if we have an array `t` whose values are $\{2, 3, 5\}$, the environment associates the value `Varr [Vint 2, Vint 3, Vint 5]` to `t`. The values of our semantics are similar to those of CompCert, except that the `Varr` constructor replaces `Vptr`, and that there is a dedicated constructor for Boolean values:

v	::=	`Vundef`	undefined value
	|	`Vbool` b	Boolean values
	|	`Vint` n | $\texttt{Vint}_{64}\ n$	32-bit and 64-bit integers
	|	$\texttt{Vfloat}_{32}\ f$ | $\texttt{Vfloat}_{64}\ f$	32-bit and 64-bit floating-point numbers
	|	`Varr` $\vec{v}$	arrays

Figure 5 gives an overview of the main rules of L_1's big-step semantics for expressions. The local environment is denoted E, while F denotes the current function. In the Eacc rule, the premise $E, F \vdash x \cdot \vec{q} \Rightarrow_P \ell$ corresponds to the evaluation of the syntactic path $x \cdot \vec{q}$ into a semantic path ℓ. Semantic paths are similar to syntactic paths except for their indices which are a single integer rather than a list of expressions. Consequently, they represent the access to a dimension-1 array. Hence, the result of the evaluation $x \cdot \vec{q} \Rightarrow_P \ell$ depends on the size variables associated to x, if any, which can be found in F. During the evaluation of a path, its validity (*i.e.*, each index is smaller than the size of the corresponding dimension) is also checked. If this check fails, then the evaluation produces an `error`, as shown by the EaccErr rule.

Figure 5 also presents some of the rules for signed division and casts between 64-bit floating-point numbers and signed 32-bit integers. They check the validity conditions for their evaluation and, in the case they are not satisfied, raise an error, which is then propagated. There are many other rules, not presented here, which cover all the arithmetic operators, casts, and constants.

As for the statements, their semantics is highly inspired by CompCert's transition semantics [Leroy 2009a], except for the memory model. Our transition semantics for L_1 manipulates the program states presented in Figure 6. Regular states correspond to the execution of the program inside the body of function F. They express that the statement s will be executed in the local environment E, and the continuation k will be executed right after. Call states express that the execution enters function F with argument values $\vec{v}$. Return states express that the execution is about to leave function F, whose environment was E, with a returned value v.

In that case, the continuation $\texttt{returnto}(y, E', F', m, k)$ expresses that the caller was function F' whose environment and continuation right before the call were E' and k, respectively. The continuation also indicates that the returned value will be written in the variable y, if it is specified. The set m contains all the arrays passed as mutable arguments. It tells how to update the

$$\textsc{Eacc}\ \frac{E, F \vdash x \cdot \vec{q} \Rightarrow_P \ell \quad E[\ell] = v \quad \texttt{primitive\_value}(v)}{E, F \vdash x \cdot \vec{q} \Rightarrow v} \qquad \textsc{EaccErr}\ \frac{E, F \vdash x \cdot \vec{q} \Rightarrow_P \texttt{error}}{E, F \vdash x \cdot \vec{q} \Rightarrow \texttt{error}}$$

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

$$\textsc{EdivsIntErr}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{Vint}\ i_1 \quad E, F \vdash e_2 \Rightarrow \texttt{Vint}\ i_2 \quad i_2 = 0 \vee (i_1 = \texttt{min\_sint} \wedge i_2 = -1)}{E, F \vdash \mathrm{divs}(e_1, e_2) \Rightarrow \texttt{error}}$$

$$\textsc{EdivsErr1}\ \frac{E, F \vdash e_1 \Rightarrow \texttt{error}}{E, F \vdash \mathrm{divs}(e_1, e_2) \Rightarrow \texttt{error}} \qquad \textsc{EdivsErr2}\ \frac{E, F \vdash e_1 \Rightarrow v_1 \quad E, F \vdash e_2 \Rightarrow \texttt{error}}{E, F \vdash \mathrm{divs}(e_1, e_2) \Rightarrow \texttt{error}}$$

$$\textsc{EcastSInt32F64}\ \frac{E, F \vdash e \Rightarrow \texttt{Vint}\ n}{E, F \vdash (\texttt{int}_{32,\texttt{Signed}} \rightarrow \texttt{float}_{64}) e \Rightarrow \texttt{Vfloat}_{64}\ f_n}$$

$$\textsc{EcastF64SInt32}\ \frac{E, F \vdash e \Rightarrow \texttt{Vfloat}_{64}\ f \quad -2^{31} - 1 < f < 2^{31}}{E, F \vdash (\texttt{float}_{64} \rightarrow \texttt{int}_{32,\texttt{Signed}}) e \Rightarrow \texttt{Vint}\ n_f}$$

$$\textsc{EcastF64SInt32Err}\ \frac{E, F \vdash e \Rightarrow \texttt{Vfloat}_{64}\ f \quad f \leq -2^{31} - 1 \vee f \geq 2^{31}}{E, F \vdash (\texttt{float}_{64} \rightarrow \texttt{int}_{32,\texttt{Signed}}) e \Rightarrow \texttt{error}}$$

Fig. 5. Semantics of L_1's expressions

Program states:	S	::=	$\mathcal{S}(E, F, s, k)$	regular state
		|	$\mathcal{C}(F, \vec{v}, k)$	call state
		|	$\mathcal{R}(E, F, v, k)$	return state
Continuations:	k	::=	`stop`	initial continuation
		|	`seq`(s, k)	continues to s, then k
		|	`loop`(s, k)	continues to `loop {`s`}`, and breaks to k
		|	`returnto`$(id^?, E, F, m, k)$	returns to the caller

Fig. 6. Program states and continuations of L_1

environment E' with the values found in the environment of the callee. If F is the main function, then the continuation is not `returnto` but `stop`, as there is no caller to return to.

The function object F contains various static data. The map `tenv` (later denoted Γ_F) assigns a type to each identifier. The map `szenv` (later denoted Σ_F) assigns size expressions to identifiers. They are organized as lists of lists of expressions to match the structure of the value. For instance, `szenv` associates `[[2 * n, m], [p + 1]]` to a $2n \times m$ matrix of arrays of size $p + 1$. The map `szenv'` (later denoted Σ'_F) has the same structure as `szenv` but it contains fresh identifiers instead of expressions. For instance, `szenv'` associates `[[j1, j2], [j3]]` to the previous matrix of arrays. Then, E`[j1]` will contain the value obtained from the evaluation of `2n` at the creation of the array. The map `penv` (later denoted ρ_F) assigns a permission to each identifier. Finally, the function object carries various invariants that ensure it is well-formed. For instance, one of these invariants states that all the size

variables are shared, and thus it is forbidden to write into them. Another invariant enforces that the return type is primitive.

$$
\begin{array}{llll}
F \quad ::= \{ & \texttt{sig} & = \{\texttt{args} = \vec{\tau}; \texttt{res} = \tau\} & \text{signature} \\
& \texttt{params} & = \vec{id} & \text{parameters} \\
& \texttt{vars} & = \vec{id} & \text{local variables} \\
& \texttt{tenv} & = id \rightharpoonup \tau & \text{variable types } (\Gamma_F) \\
& \texttt{szenv} & = id \rightharpoonup [[\vec{e}], \ldots] & \text{size expressions } (\Sigma_F) \\
& \texttt{szenv'} & = id \rightharpoonup [[\vec{id}], \ldots] & \text{size variables } (\Sigma'_F) \\
& \texttt{penv} & = id \rightharpoonup \{\texttt{Shared}, \texttt{Mutable}, \texttt{Owned}\} & \text{permissions } (\rho_F) \\
& \texttt{body} & = s \quad \} &
\end{array}
$$

`Mutable` is the permission associated to variables annotated with the `mut` keyword. `Owned` is associated to local variables and arrays that can be dynamically allocated and freed. All the other variables have the `Shared` permission. During function calls, it is possible to lower the permission of a passed array, according to the total order `Shared` < `Mutable` < `Owned`.

A selection of semantics rules for L_1's statements is shown in Figure 7. The skipped rules, which deal with control flow, are very classic and similar to the corresponding ones in C#minor. In the definition of the rules, G denotes the global environment which contains all the function declarations. Notice that the notion of trace is missing from our semantics rules, thus deviating from CompCert. These traces track observable behaviors like external calls, and CompCert proves that these traces are preserved across compilation passes. We currently have no use for them, but we might bring them back later.

The rule Write that modifies the environment is quite similar to the rule Eacc for reading it, though it performs a few more checks. First, it checks that the permission of the written location is at least `Mutable`. It also checks that the stored value is well-typed with respect to the type associated to the semantic path ℓ in Γ_F. This leak of typing information is necessary for the semantics preservation of L_2 to C#minor because we must ensure that we write as many bytes as the type $\Gamma_F(\ell)$ states.

The rules Alloc and Free deal with dynamic allocations of arrays. Only dimension-1 arrays are currently supported. For an allocation whose size is an expression s that evaluates to a value n, a size variable i is added to the environment to keep track of this dynamic size n. The array added to the environment is initialized with the default value v for the type τ, which is usually zero. To free an array stored in a variable x, the most important premises are that the permission of x is `Owned` and that x is actually part of the environment. It is then removed from the environment, so freeing it a second time or accessing it is undefined behavior.

The rule CallInternal handles the call to an internal function, that is, a function written in our language. It evaluates the arguments and checks that their values are well-typed with respect to the signature of the callee. It also checks that the permissions of the passed arguments are at least the ones requested by the callee. It also checks that types are the same in the typing environment of both caller and callee. The `valid_call` predicate ensures that the values passed as size arguments are equal to the sizes of the corresponding arrays in the caller. Finally, the last premise checks that arrays passed as mutable arguments do not alias with any other argument, by comparing their paths ($\ell_i \preceq \ell_j$ means "ℓ_i is a prefix of ℓ_j").

There are two error rules for the function call. If an error is raised during argument evaluation, the first rule propagates the error. The second error rule is quite similar to the CallInternal rule, but it deals with the case where the premise $\texttt{valid\_call}(E, F, \texttt{Internal}(F'), \vec{p})$ fails.

The rule ReturnState restores the mutable arrays by updating the environment of the caller. It uses the set m, built by the CallInternal rule, which associates each path to a mutable array with

$$\textsc{Write}\ \frac{\begin{array}{c} E,F \vdash e \Rightarrow v \qquad \texttt{primitive\_value}(v) \\ \rho_F(x) \geq \texttt{Mutable} \qquad E,F \vdash x \cdot \vec{q} \Rightarrow_P \ell \\ \Gamma_F(\ell) = \tau \qquad v \in \tau \end{array}}{G \vdash \mathcal{S}(E,F,x \cdot \vec{q} \leftarrow e,k) \rightarrow \mathcal{S}(E[\ell \leftarrow v],F,\texttt{skip},k)}$$

$$\textsc{WriteErrVal}\ \frac{E,F \vdash e \Rightarrow \texttt{error}}{G \vdash \mathcal{S}(E,F,x \cdot \vec{q} \leftarrow e,k) \rightarrow \mathcal{S}(E,F,\texttt{error},k)}$$

$$\textsc{WriteErr}\ \frac{\begin{array}{c} E,F \vdash e \Rightarrow v \qquad \texttt{primitive\_value}(v) \\ \rho_F(x) \geq \texttt{Mutable} \qquad E,F \vdash x \cdot \vec{q} \Rightarrow_P \texttt{error} \end{array}}{G \vdash \mathcal{S}(E,F,x \cdot \vec{q} \leftarrow e,k) \rightarrow \mathcal{S}(E,F,\texttt{error},k)}$$

$$\textsc{Alloc}\ \frac{\begin{array}{c} \Gamma_F(x) = \tau\ \texttt{arr} \qquad \Sigma_F(x) = [[s]] \qquad \Sigma'_F(x) = [[i]] \qquad \rho_F(x) = \texttt{Owned} \\ \texttt{default\_value}(\tau) = v \qquad E,F \vdash s \Rightarrow \texttt{Vint}_{64}\ n \qquad n \times \texttt{sizeof}(\tau) \leq \texttt{max\_uint64} \end{array}}{G \vdash \mathcal{S}(E,F,x \leftarrow \texttt{alloc},k) \rightarrow \mathcal{S}(E[i \leftarrow \texttt{Vint}_{64}\ n][x \leftarrow \texttt{Varr}\ (\texttt{repeat}(n,v))],F,\texttt{skip},k)}$$

$$\textsc{Free}\ \frac{\begin{array}{c} \Gamma_F(x) = \tau\ \texttt{arr} \qquad \Sigma_F(x) = [[s]] \qquad \Sigma'_F(x) = [[i]] \qquad \rho_F(x) = \texttt{Owned} \\ E[x] = \texttt{Varr}\ \vec{v} \end{array}}{G \vdash \mathcal{S}(E,F,\texttt{free}\ x,k) \rightarrow \mathcal{S}(E \setminus \{x,i\},F,\texttt{skip},k)}$$

$$\textsc{CallInternal}\ \frac{\begin{array}{c} G(f) = \texttt{Internal}(F') \qquad |\vec{p}| = |F'.\texttt{sig.args}| \qquad \vec{x} = F'.\texttt{params} \\ \forall i,\ E,F \vdash p_i \Rightarrow_P \ell_i \qquad \forall i, E(\ell_i) = v_i \qquad \vec{v} \in F'.\texttt{sig.args} \\ \forall i, \rho_F(\ell_i) \geq \rho_{F'}(x_i) \qquad \forall i, \Gamma_F(\ell_i) = \Gamma_{F'}(x_i) \\ \texttt{valid\_call}(E,F,\texttt{Internal}(F'),\vec{a}) \\ \forall i\ j,\ i \neq j \wedge \rho_{F'}(x_i) \geq \texttt{Mutable} \Rightarrow \ell_i \not\leq \ell_j \wedge \ell_j \not\leq \ell_i \\ m = \{(\ell_i,x_i) \mid \rho_{F'}(x_i) \geq \texttt{Mutable}\} \end{array}}{G \vdash \mathcal{S}(E,F,y \leftarrow f(\vec{p}),k) \rightarrow \mathcal{C}(F',\vec{v},\texttt{returnto}(y,E,F,m,k))}$$

$$\textsc{CallState}\ \frac{}{G \vdash \mathcal{C}(F,\vec{v},k) \rightarrow \mathcal{S}(\texttt{build\_env}(F.\texttt{params},\vec{v}),F,F.\texttt{body},k)}$$

$$\textsc{Return}\ \frac{E,F \vdash e \Rightarrow v \qquad v \in F.\texttt{sig.sig\_res}}{G \vdash \mathcal{S}(E,F,\texttt{return}\ e,k) \rightarrow \mathcal{R}(E,F,v,\texttt{destructCont}(k))}$$

$$\textsc{ReturnState}\ \frac{\begin{array}{c} \forall (\ell,x) \in m,\ E[p] = \texttt{Varr}\ \_ \ \wedge\ E'[x] = \texttt{Varr}\ \_ \\ \forall (\ell,x) \in m,\ \Gamma_F[\ell] = \Gamma_{F'}[x] \\ \texttt{primitive\_value}(v) \qquad E_{upd} = \texttt{update\_env}(E,m,E') \end{array}}{G \vdash \mathcal{R}(E',F',v,\texttt{returnto}(y,E,F,m,k)) \rightarrow \mathcal{S}(E_{upd}[y \leftarrow v],F,\texttt{skip},k)}$$

$$\textsc{Error}\ \frac{}{G \vdash \mathcal{S}(E,F,\texttt{error},k) \rightarrow \mathcal{S}(E,F,\texttt{error},k)}$$

Fig. 7. Transition semantics for L_1's statements

the corresponding parameter in the called function. To avoid any creation of alias, we forbid the returned value to be an array.

4.2 Typing

For the language to be safe, no well-typed program should trigger an undefined behavior. Our semantics currently has two major sources of undefined behavior: access to an uninitialized variable (whose value is thus missing from the execution environment) and aliasing of a mutable array with some other arguments during function call. Figure 8 presents the typing rules of L_1 for statements, as they are the most interesting ones regarding safety. A typing sequent $G, F, U \vdash s : (U_n, U_c, U_b, U_r)$ states that the statement s is well-typed when execution starts with a set U of potentially uninitialized variables and ends with a set U_n. The three other sets U_c, U_b, and U_r describe the status of the variables when the statement abruptly exits with `continue`, `break`, and `return`, respectively. Sets of potentially uninitialized variables are set to empty when the corresponding outcome is impossible.

Since either branch of a conditional can *a priori* be evaluated, the typing rule for the conditional takes the union of the sets of uninitialized variables of both branches for each outcome. In the typing rules for the sequence, `abrupt_ending`(s_1) holds if and only if the only outcomes of statement s_1 are `continue` and `return`. As for the typing rule of the loop, notice that it checks that the set U is actually a fixed point across the iterations.

In the typing rule for function call, several premises are quite similar to the ones of the semantic rule CallInternal. The second premise checks that all the size variables are initialized. Note that the rule forbids parameters to be declared as `Owned`, since ownership transfer is not fully implemented yet. The last premise checks the absence of aliasing. The only difference with the corresponding premise of CallInternal is that the semantic paths are here replaced with the syntactic paths $\vec{p}$. Consequently, $p_i \preceq p_j$ is still the prefix operator but in a fuzzier way. Indeed, when it compares subexpressions pairwise, it considers them to be potentially equal if either of them accesses a variable. For instance, `x + 1` and `5` are potentially equal, but `(u64)(2.5 / 2.0) * 4` and `1` are certainly not. As with overestimated sets of uninitialized variables, this rule can cause programs to be rejected by the type checker, despite having no undefined behaviors at runtime.

5 FORMAL VERIFICATION

We now want to formally ensure some guarantees about our language and its compiler. The first one is the type safety of our language, that is, a well-typed program has no undefined behavior with respect to L_1 semantics. This ensures, in particular, that function calls do not create any alias. Section 5.1 describes the type safety theorem and its proof.

The other guarantee we want is that no miscompilation occurred, that is, compiled programs behave the same as source programs. This property is usually proved using backward simulation, like CompCert's final theorem. Forward simulation, however, is often easier to prove and, if the output language is deterministic, it implies backward simulation [Leroy 2009a]. Therefore, we prove forward simulation between L_1 and C#minor, as described in Section 5.2.

The latter result is then composed with CompCert's result and we obtain a semantics preservation theorem for the whole compiler. Together with type safety, we can now state that the execution of the generated code is safe.

5.1 Type safety

Type safety relies on some invariants to ensure the consistency of the semantics states during the execution. For a regular state $\mathcal{S}(E, F, s, k)$, the invariants are as follows:

1. E is well-typed with respect to Γ_F.
2. If an array t is defined (*i.e.*, $t \notin U$), then all its size variables are defined.

$$\frac{}{G, F, U \vdash \texttt{skip} : (U, \emptyset, \emptyset, \emptyset)} \qquad \frac{\begin{array}{c} x \notin U \vee \vec{q} = [] \qquad \Gamma_F(x \cdot \vec{q}) = \tau \qquad \Sigma_F(x) = S \\ \forall i\ j,\ |S_i| = |q_i| \wedge \Gamma_F(q_{i,j}) = \texttt{int}_{64,\texttt{Unsigned}} \\ F, U \vdash e : \tau \qquad \texttt{primitive\_type}(\tau) \qquad \rho_F(x) \geq \texttt{Mutable} \end{array}}{G, F, U \vdash x \cdot \vec{q} \leftarrow e : (U \setminus \{x\}, \emptyset, \emptyset, \emptyset)}$$

$$\frac{\begin{array}{c} \Gamma_F(x) = \tau\ \texttt{arr} \qquad \Sigma_F(x) = [[s]] \qquad \Sigma'_F(x) = [[i]] \qquad \rho_F(x) = \texttt{Owned} \\ F, U \vdash s : \texttt{int}_{64,\texttt{Unsigned}} \qquad i \in U \qquad \texttt{primitive\_type}(\tau) \qquad \tau \neq \texttt{void} \end{array}}{G, F, U \vdash x \leftarrow \texttt{alloc} : (U \setminus \{x, i\}, \emptyset, \emptyset, \emptyset)}$$

$$\frac{\begin{array}{c} \Gamma_F(x) = \tau\ \texttt{arr} \qquad \Sigma_F(x) = [[s]] \qquad \Sigma'_F(x) = [[i]] \qquad \rho_F(x) = \texttt{Owned} \\ x \notin U \qquad \texttt{primitive\_type}(\tau) \end{array}}{G, F, U \vdash \texttt{free}\ x : (U \cup \{x, i\}, \emptyset, \emptyset, \emptyset)}$$

$$\frac{\begin{array}{c} G(f) = F' \qquad \forall i,\ F, U \vdash p_i : F'.\texttt{sig.args}_i \qquad \Gamma_F(y) = F'.\texttt{sig.res} \\ \forall i\ j\ k, (\Sigma_F(p_k))_{i,j} \notin U \qquad \vec{x} = F'.\texttt{params} \\ \forall i, \rho_{F'}(x_i) \neq \texttt{Owned} \qquad \forall i, \rho_F(p_i) \geq P_{F'}(x_i) \\ \forall i, \rho_{F'}(x_i) \geq \texttt{Mutable} \Rightarrow \Gamma_{F'}(x_i) = \texttt{arr} \\ \forall i\ j, i \neq j \wedge \rho_{F'}(x_i) \geq \texttt{Mutable} \Rightarrow p_i \not\leq p_j \wedge p_j \not\leq p_i \end{array}}{G, F, U \vdash y \leftarrow f(\vec{p}) : (U \setminus \{y\}, \emptyset, \emptyset, \emptyset)}$$

$$\frac{F, U \vdash e : F.\texttt{sig.res}}{G, F, U \vdash \texttt{return}\ e : (\emptyset, \emptyset, \emptyset, U)} \qquad \frac{F, U \vdash e : \texttt{bool}}{G, F, U \vdash \texttt{assert}\ e : (U, \emptyset, \emptyset, \emptyset)}$$

$$\frac{F, U \vdash e : \texttt{bool} \qquad G, F, U \vdash s_1 : (U_n, U_c, U_b, U_r) \qquad G, F, U \vdash s_2 : (U'_n, U'_c, U'_b, U'_r)}{G, F, U \vdash \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\} : (U_n \cup U'_n, U_c \cup U'_c, U_b \cup U'_b, U_r \cup U'_r)}$$

$$\frac{\begin{array}{c} G, F, U \vdash s_1 : (U_n, U_c, U_b, U_r) \\ \texttt{abrupt\_ending}(s_1) \end{array}}{G, F, U \vdash s_1; s_2 : (U_n, U_c, U_b, U_r)} \qquad \frac{\begin{array}{c} G, F, U \vdash s_1 : (U_n, U_c, U_b, U_r) \\ \neg\texttt{abrupt\_ending}(s_1) \\ G, F, U_n \vdash s_2 : (U'_n, U'_c, U'_b, U'_r) \end{array}}{G, F, U \vdash s_1; s_2 : (U'_n, U_c \cup U'_c, U_b \cup U'_b, U_r \cup U'_r)}$$

$$\frac{\begin{array}{c} G, F, U \vdash s : (U_n, U_c, U_b, U_r) \\ U_n \subseteq U \qquad U_c \subseteq U \end{array}}{G, F, U \vdash \texttt{loop}\{s\} : (U_b, \emptyset, \emptyset, U_r)} \qquad \frac{}{G, F, U \vdash \texttt{break} : (\emptyset, \emptyset, U, \emptyset)}$$

$$\frac{}{G, F, U \vdash \texttt{continue} : (\emptyset, U, \emptyset, \emptyset)} \qquad \frac{}{G, F, U \vdash \texttt{error} : (\emptyset, \emptyset, \emptyset, \emptyset)}$$

Fig. 8. Typing rules for L_1's statements

3. Arrays sizes are valid, that is, their product, for multidimensional arrays, is smaller than the actual size of their corresponding arrays values (*i.e.*, the number of elements in their `Varr` value).
4. The statement s is well-typed.
5. s is well-formed, that is, it does not contain any `break` or `continue` instruction unless the continuation states that this instruction is in a loop (*i.e.*, it contains a `Kloop` before the topmost `Kreturnto`).

6. The continuation k is well-typed and maintains numerous invariants. Most of these invariants are related to function calls. For instance, sizes and shapes of arrays passed as arguments are the same in both callers and callees.

This gives rise to the following theorem about type safety:

Theorem 5.1 (Type safety). *Let p be a L_1 program and s a state of its semantics. If p is well-typed, s is not a final state, and all the invariants hold in s, then there exists a state t such that $s \longrightarrow t$ and all the invariants hold in t.*

The well-typedness of the program p in this theorem corresponds to the result of the typechecking algorithm that is actually implemented in the compiler. As a consequence, the theorem not only implies the safety of the language, but it also shows that the implemented typechecker is correct.

Concerning the proof, while being quite long, it is mostly straightforward. Indeed, most of the safety issues related to memory accesses will be handled during the proof of semantics preservation between L_2 to C#minor. The only remaining issue about memory accesses is to prove that, if the type checker has not detected any alias during function call, there cannot be any alias at execution, which is trivial.

While the proof is overall straightforward, that does not mean it is simple. One of the main difficulties comes from the large number of invariants we must maintain. Another important difficulty in this proof arises from the tracking of uninitialized variables shown in Section 4.2. This makes the invariants quite subtle, since some of them are valid only for initialized variables. This issue has also interfered a lot with the tracking of dead code.

5.2 Semantics preservation

Concerning compiler correctness, the theorem we want to show is the following, where $\Leftrightarrow$ denotes a relation between the states L_1 and C#minor.

Theorem 5.2 (Semantics preservation by forward simulation). *Let s and t be two states of L_1's semantics. Let s' be a state of C#minor's semantics. If the source program progresses from s to t and if $s \Leftrightarrow s'$, then there exists a C#minor's state t' such that the C#minor generated program progresses from s' to t' and $t \Leftrightarrow t'$.*

The semantics preservation theorem is proved separately for each pass between L_1 and C#minor. This simplifies the proof a bit, by separating the generation of dynamic tests from the implementation of the copy-restore semantics. In each proof, there is a matching predicate that links semantics states of the corresponding input and output languages.

The main difficulty for the correctness of the first pass is to prove that the generated dynamic tests are both necessary and sufficient. This is especially tedious for casts between primitive types. For casts from floating-point numbers to integers, we have to prove that tests evaluate to true if and only if the floating-point number is in the bounds of representable integers. For instance, the cast (`f32` $\longrightarrow$ `u64`) `f` generates the test `-1 < f < 2^64` which is unexpectedly difficult to prove, due to the presence of the necessary ranges $(-1, 0)$ and $(2^{64} - 1, 2^{64})$. Finally, the fact that the added assertions should not cause any undefined behavior themselves makes the proof a bit cumbersome. This issue is magnified by the fact that the `guarded` function optimizes redundant tests away.

The matching predicate between the semantics states of L_2 and C#minor mostly deals with the relation between our local environments and C#minor's memory model. In fact, not only does this relation exist for the local environment of the current function, but it also exists for the environments of functions further in the call stack (*i.e.*, in the continuation). To represent it, we introduce a translation function $t : \mathbb{N} \times \mathbb{P} \to \mathbb{N}^* \times \mathbb{N} \times \mathbb{V}$, where $\mathbb{P}$ is the set of semantics paths and $\mathbb{V}$ is $\{\texttt{Visible}\} \cup \{\texttt{Hidden}(\ell) \mid \ell \in \mathbb{P}\}$. The equality $t(n, \ell) = (b, o, \texttt{Visible})$ states that the value

corresponding to path ℓ in the environment E_n at level n (functions are called with increasing levels and the `main` function is at level 0) is located in C#minor's memory at offset o in block b. The invariant between L_2's environments E_n and C#minor's memory M is stated as follows:

$$\begin{array}{lll} \forall n\ \ell\ \vec{v}, & E_n[\ell] = \texttt{Varr}\ \vec{v} \Rightarrow & \\ & \exists b\ o\ s, \quad t(n,\ell) = (b,o,s) \wedge & \\ & \qquad s = \texttt{Visible} \Rightarrow & \\ & \qquad M[(b, -\texttt{sizeof(ptr)})] = \texttt{sizeof}(\Gamma_{F_n}[\ell +\!\!+ [0]]) \times |\vec{v}| \wedge & \\ & \qquad \left(\begin{array}{ll} \forall o, & -\texttt{sizeof(ptr)} \le o < \texttt{sizeof}(\Gamma_{F_n}[\ell +\!\!+ [0]]) \times |\vec{v}| \Rightarrow \\ & M[(b,o)] \text{ freeable} \end{array}\right) \wedge & \\ & \qquad \left(\begin{array}{ll} \forall i, & 0 \le i < |\vec{v}| \Rightarrow \\ & M[(b, o + i \times \texttt{sizeof}(\Gamma_{F_n}[\ell +\!\!+ [0]]))] = \texttt{transl\_value}(v[i]) \end{array}\right) & \end{array} \tag{1}$$

The first two consequents of this invariant ensure that the requirements of CompCert's for memory deallocation will be met, and thus C#minor's semantics will progress. The third consequent relates the values in L_2's environments and those in C#minor's memory M.

The subtlety of this invariant lies in the antecedent $s = \texttt{Visible}$. Indeed, it is possible for two different paths, at different levels, to point to the same memory area. This is the case for arrays passed as mutable arguments. Consequently, such mutable arrays in the caller function must be ignored until the execution returns to it. Therefore, we have introduced a notion of path visibility. A visible path has not been passed as mutable to a called function, while a hidden one has. The equality $t(n,\ell) = (\_, \_, \texttt{Hidden}(\ell'))$ means that the path ℓ, at level n, is hidden by the path ℓ' at level $n+1$. When the function returns, the value of $E_{n+1}[\ell']$ will be written into $E_n[\ell]$. The `returnto` continuations also maintain a relation between the hidden paths and the set m of arrays passed as mutable arguments.

Moreover, some well-formedness invariants on t also need to be preserved. One of these invariants states that, if a path ℓ was associated to offset o at block b, then, if we pass ℓ as argument, it will still be linked to offset o at block b in the callee. In other words, we actually pass arrays by reference.

$$\forall n\ \ell\ \ell'\ b\ o, t(n,\ell) = (b,o,\texttt{Hidden}(\ell')) \longrightarrow t(n+1,\ell') = (b,o,\_) \tag{2}$$

On function return, hidden paths (*i.e.*, passed as mutable arguments) are restored from E_{n+1} to E_n. We can clearly state that any array that was not passed as mutable is not modified in E_n. Such a statement, however, is far from obvious on the side of C#minor's memory M. It is needed in order to prove that invariant (1) is preserved. As a consequence, we need to ensure that arrays passed as mutable arguments are separated from all the other arrays in M, which can be phrased as: function t associates them to separate memory blocks. In order to prove this property, we need to preserve the following invariants during the whole semantics preservation proof, where F_n denotes the function at level n of the call stack.

$$\begin{array}{ll} \forall n\ \ell\ b\ o, & t(n,\ell) = (b,o,\texttt{Visible}) \wedge \rho_{F_n}(\ell) \ge \texttt{Mutable} \Rightarrow \\ & \forall \ell'\ b'\ o', \ell \ne \ell' \wedge t(n,\ell') = (b',o',\_))) \Rightarrow b \ne b' \end{array} \tag{3}$$

$$\begin{array}{ll} \forall n\ \ell\ b\ o, & t(n,\ell) = (b,o,\texttt{Visible}) \wedge \rho_{F_n}(\ell) \ge \texttt{Mutable} \Rightarrow \\ & \forall m\ \ell'\ b'\ o', m < n \wedge t(m,\ell') = (b',o',\texttt{Visible}) \Rightarrow b \ne b' \end{array} \tag{4}$$

Invariant (3) states that, at any level n, each visible path ℓ, which is at least mutable, is separated from all other paths ℓ', no matter their visibility. In other words, $t(n,\ell)$ and $t(n,\ell')$ point to different memory blocks. This invariant is a consequence of the non-aliasing premise of the semantics rule CallInternal. It entails that writing into a mutable array does not modify the block associated with any other array that can be accessed at the same level n.

Table 2. Size of the development in number of lines

	Code	Specification	Proof
Parser, typing, and simplifications (OCaml)	2128	–	–
Syntax, types, and typing	1472	–	393
Common semantics definitions and proofs	–	1531	1160
Semantics of L_1	–	885	496
Semantics of L_2	–	368	108
Type safety	–	1112	3076
Compilation from L_1 to L_2	322	577	1524
Compilation from L_2 to C#minor	562	1521	3171
Miscellaneous	–	979	936
Total	4484	6973	10864

Invariant (4), which also deals with path ℓ at level n, is more subtle. It states that ℓ has to be separated from any other visible path ℓ' at any level $m < n$. Indeed, if path ℓ is visible at level $n > m$, then, once the function returns to level m, there might exist a path ℓ_0 that is a prefix of ℓ. Thus, modifications made on $E_n[\ell]$ will be propagated to $E_m[\ell_0]$. As for the path ℓ' at level m, it has not been passed as mutable to a higher level, since its visibility is not `Hidden`. As a consequence, invariant (1) states that its associated memory block has to contain the same values as $E_m[\ell']$. The latter is impossible if function t linked ℓ' and ℓ_0 to the same memory blocks. Therefore, since ℓ and ℓ_0 are in the same block, which is a consequence of invariant (2), we need to ensure that ℓ and ℓ' point to different blocks through t.

5.3 Formal proofs

The semantics of L_1 and L_2, along with the proofs of semantics preservation and type safety, have been formalized in the Coq proof assistant. Just as with CompCert, this implies thousands of lines for both specification and proofs. We sum this up in Table 2. Unless otherwise specified, these are lines of Coq code.

Using a proof assistant for this kind of proof was very useful to find errors in typing, translations, and semantics. Many parts of a compiler can be wrong, starting with the compilation passes. Indeed, it is hard to keep track of every case and it is easy to introduce undefined behaviors during a translation. For instance, during the translation from L_1 to L_2, if we had messed the optimization of assertions, it would have introduced some undefined behaviors that would have been difficult to detect by testing. Writing the assertions themselves was also error-prone. Indeed, the validity range of the casts from floating-point numbers to integers was a bit subtle. For instance, to guard a cast from a 64-bit floating-point number to a 64-bit unsigned integer, a seemingly correct test is $0 \leq f < 2^{64} - 1$. It is indeed safe, as it prevents undefined behavior from happening, but it is not correct, as some valid programs would spuriously raise a runtime error.

Typing rules were also a large source of errors and our first version was far different from the one shown on Figure 8. The safety proof forced us to change a lot of things in the typing of statements. For instance, the premise for the assignment $x \cdot \vec{q} \leftarrow e$ was originally just $x \notin U$ rather than $x \notin U \vee \vec{q} = [\,]$. As a consequence, it was impossible to directly write into local variables to initialize them, but the bug went unnoticed because local variables were wrongly assumed to be initialized at function entry.

Last but not least, some subtle mistakes were introduced in the semantics, thus preventing it from matching the implemented compiler or, even worse, it could be unrealizable. Indeed, when writing

a compiler, one assumes some semantics for the destination language. However, if this assumed semantics is different from the real one, the compiler would seem correct while it is not. It is only once the gap with a lower-level semantics is formally bridged that the issue becomes apparent. For example, we were under the impression that local variables ($x \cdot \vec{q}$ with $\vec{q} = []$) and array cells ($\vec{q} \neq []$) were supporting the same kind of operations, so the semantics was handling them interchangeably. But as far as CompCert is concerned, while the program can perform byte-level accesses through pointers, only register-sized manipulations are allowed on local variables. This issue became apparent once we tried to prove the semantics preservation between L_2 and C#minor.

Finally, while the benefits of the formal verification are large, it also requires a large work. In particular, proving all those theorems in an interactive way was a bit tedious. Indeed, some proof tactics, especially those used in the safety proof, take a long time to check because a lot of cases need to be processed at the same time. Moreover, CompCert's style of semantics, which uses a lot of huge inductive predicates, leads to hypothesis naming problems, which makes the proof process quite painful, as the proof scripts become rather fragile with respect to specification changes.

6 RELATED WORKS

The WhyMP library provides a formally verified implementation of numerous algorithms from the GMP library [Melquiond and Rieu-Helft 2023; Rieu-Helft 2020]. These algorithms were first implemented using the WhyML language, then formally verified using the Why3 platform, and finally extracted to C code. This formalization is based on a WhyML model of a subset of the C language that is expressive enough to implement GMP's algorithms. As such, this was a major source of inspiration for the design of our language. But contrarily to our work, that WhyML model of C, which includes a memory model and a compilation scheme, was never verified in any way and might well be inconsistent. The most important feature we are currently missing is the ability to partition a mutable array into two non-aliased arrays.

When it comes to handling aliasing, the type system of the Rust language was also an inspiration. But since our type system is much simpler and avoids any notion of lifetime, an approach slightly closer to ours comes from the SPARK subset of Ada [Jaloyan et al. 2020]. Again, there was no formal proof of the consistency of their model.

Regarding Rust, there have been several works toward the formalization of the language [Jung et al. 2019, 2017]. A similarity with our work is that they disregard as much as possible the type system, so that they can also reason about unsafe programs, that is, untyped ones. As with ours, it is their semantics that take care of anything related to aliasing. Their semantics, however, is much more dynamic than ours, up to the point that they have an interpreter that can run Rust programs and detect undefined behaviors. These works on Rust focus on the language and its semantics and do not touch the matter of formally verifying the compiler in any way.

We went the way of defining a safe language on top of C, but another approach could have been to let the user prove the safety of their C programs. This is the path followed by VST [Appel 2011; Cao et al. 2018]. It provides a separation logic for the Cminor language of CompCert, as well as various tools to help users formally prove the Hoare triples of their programs using Coq. Thanks to VST being built on top of CompCert, once a source program has been verified with VST, its functional correctness is preserved till the Assembly code. We are currently lacking a program logic for our language that would let us talk about functional correctness rather than just safety.

On the topic of building a safe language on top of C, another initiative is the Checked C language [Elliott et al. 2018; Li et al. 2022; Ruef et al. 2019]. The safe part of the language is mostly based on rich pointer types, and the compiler introduces bound variables for pointers that are the target of arithmetic operations to track the extent of accessible addresses. A semantics for

this language was formalized in Coq, as well as a proof of its safety, in particular the fact that, if a program mixes both C and Checked C, any undefined behavior necessarily comes from the C part.

CompCert is not the only formally verified compiler. Among generalist compilers, the other system of note is CakeML [Kumar et al. 2014; Myreen 2021], which has been verified using the HOL4 theorem prover. As a compiler, the main difference with CompCert is that its semantics preservation theorem goes until the machine code rather than stopping at Assembly. But its input language, in the family of Standard ML, as well as its semantics, makes it much less suited than CompCert for the kind of low-level libraries we target.

While building on top of CompCert makes it possible to have a formally verified compiler from end to end, it is a hindrance when it comes to generating high-performance code, as it performs relatively few optimizations. A potential solution would be to build on top of the Chamois fork of CompCert, which provides various loop optimizations as well as strength reduction [Gourdin et al. 2023]. Another possibility would be to branch out of CompCert after the first few passes of the compiler and to use a different backend to complete the compilation. For example, one could plug into the LLVM compiler at the level of LLVM IR. To do so, one would need to convert the program to an SSA form, which can be achieved by using the CompCert SSA variant of the compiler [Barthe et al. 2014], and then to translate it to LLVM IR, which has been formalized in Coq through the Vellvm project [Zhao et al. 2012].

Finally, there is a wide variety of works that focus on designing programming languages to manipulate arrays. In particular, one can find various type systems based on dependent types to encode the shapes and sizes of arrays [Colaço et al. 2023; Henriksen and Elsman 2021]. If not for the fact that we wanted to ensure compatibility with existing libraries such as GMP and its peculiar ordering of arguments, we would presumably have followed a similar line of thoughts and looked for a system closer to unadulterated dependent types.

7 CONCLUSION

In this article, we have presented a safe language suitable for implementing the basic blocks of computer algebra systems. Since most state-of-the-art implementations of these algorithms are written using low-level programming languages, namely C and Fortran, we have tried to stay as close as possible to those languages, while importing a non-aliasing philosophy that makes reasoning about programs simpler. That way, implementing these algorithms in our language does not suffer from a steep learning curve. Such algorithms are already sufficiently difficult to get correct on their own, and we did not want to make their implementation even harder by imposing a change of paradigm on their developers.

Since these algorithms make a heavy use of multidimensional arrays to represent mathematical objects, they are the core construction of our language. Moreover, sizes and shapes of arrays are explicit, so that one can write signatures that properly model the behavior of functions from GMP or BLAS. This makes it simple to invoke such functions in a safe way from our language, assuming that they have been faithfully modeled.

The simplicity of our language comes with a cost, though. Indeed, while we wanted to express pointer validity and non-aliasing through the type system, we did not want to have lifetime annotations à la Rust. Indeed, their expressiveness does not seem worth their complexity, for the class of algorithms we are interested in. That said, despite not having lifetimes, we were hoping to have a rather rich system of permissions and be able to manipulate mutable arrays of shared arrays, or the converse. This proved to be impossible. Consider the following function. At first sight, the assignment seems to be fine because `a[0]` and `b` have the same type, *i.e.*, a shared array of integers.

```
fun f(a: mut [[i32; n]; m], b: [i32; n]; m n: u64) { a[0] = b; }
```

Still, this operation cannot be allowed for two reasons. First, if the array passed as `b` was originally mutable, an alias to it (through the array passed as `a`) arises once the function returns. Second, if the array passed as `b` is freed before the one passed as `a`, then `a[0]` becomes a dangling pointer. In Rust, lifetimes prevent this issue by forcing the user to explicitly specify that the array passed as `a` will live at least as long as the one passed as `b`, and typing will fail otherwise. In a similar way, the absence of lifetimes means that functions from our language cannot return borrows.

7.1 The compiler

An important part of the presented work is the design of a full compiler for our language. The semantics of the language has been formalized with the Coq proof assistant along with the safety of the type system and the correctness of the compiler. Together, these mechanically checked theorems entail that the compiled code is safe and behaves the same way the source code does. The formalization effort amounts to roughly 20 person-months. The development is axiom-free, except for the assumption that external functions, whose code is unknown, do not break the invariants of type safety and semantics preservation.

Performance-wise, the choice of CompCert is hardly optimal, especially for the kind of applications we target. To address the cases where performance is more important than guarantees on generated programs, our compiler can also produce C code. This translation is unverified, so we are considering a translation of L_2 to LLVM IR, with a proof of semantics preservation based on Vellvm to increase the confidence in its correctness.

Another issue regarding performance lies in the compilation pass from L_1 to L_2, as it introduces numerous dynamic checks, especially those for out-of-bound accesses. Our frontend currently eliminates some of them, and experiments show that an optimizing compiler can eliminate most if not all the remaining ones. The main issue is that the user has currently no way of knowing whether any dynamic check remains, short of perusing through the generated Assembly code. It might be better to add a new compilation pass that removes the generated assertions using some decision procedure, *e.g.*, on linear arithmetic. If any dynamic check were to remain, it would be a compilation error, unless the user has explicitly flagged the corresponding array access as needing such a check.

7.2 Language extensions

Some important constructions have yet to be implemented in our language. First of all, dynamic allocations are currently allowed only for arrays of primitive types. Arrays of arrays have to come from the external world. On a related note, dynamic arrays are currently initialized with zeros, which is a waste of cycles if the user intends to manually initialize them with some other values. Lastly, it should be possible to transfer the ownership of dynamic allocations through function calls, rather than just borrowing them as mutable arguments.

Having disjoint mutable views on arrays is another important feature. In particular, they are needed for divide-and-conquer algorithms, such as Toom-Cook's fast multiplication, or the block matrix multiplication. Such views are provided by the memory model of C used in WhyMP and have proved to be sufficient to implement and verify many GMP algorithms [Rieu-Helft 2020]. We do not expect any major difficulty, as our language already supports splitting mutable arrays across function calls (*e.g.*, `f(t[0], t[1])` with `t` an array of arrays and `f` a function with two mutable arguments). So, it is just a matter of generalizing such splits and allowing them inside the current function. The simplest way would be through plain lexical scoping (*i.e.*, just a new kind of continuation in our semantics), but since our type system already tracks uninitialized variables in subtle ways, we might even go for non-lexical views.

Finally, the language also needs records. This should cause no difficulty, as this is just a matter of adding both a new kind of value and a new kind of path component to access their fields. As long as records are passed as pointer arguments to functions,[4] the whole formalization we currently have for arrays can be reused. We also want the possibility, for a record, to own an array whose size would be given by one of its field. This would make it possible to reimplement in our language the functions from the `mpz` layer of GMP.

7.3 Functional verification of programs

The semantics of our language has been designed from the start to simplify reasoning about programs. This results, among other things, in arguments being passed by copy-restore. As a consequence, it is easier to understand whether an array can be modified by a function call. More generally, as long as at least one path to an object starts with an identifier marked as shared, there is no need to state any kind of invariant to guarantee that its value is immutable. This considerably reduces the need for reasoning about programs using a separation logic or any related approach.

While it is already possible to reason about a program written in our language by directly looking at its small-step semantics, it is a bit tedious and does not scale. So, the next step is to devise a program logic, in order to ease the deductive verification of functional correctness. First, one would build a big-step semantics, for which CompCert already provides the needed tools, since such a work was done for Cminor. Then, one would build Hoare triples on top of it and the associated weakest precondition computation. Since our semantics has no memory model and everything lives in the local environment, this is the optimal setup for defining a program logic. Indeed, assignments in the program are just substitutions in logical propositions.

By having a formalized programming language, a formally verified compiler, a program logic, and all of that inside Coq, it even becomes sensible to run such programs directly inside the proof assistant to perform intensive computational proofs. In order to trust such a proof, one would mostly have to trust that Coq and the computing environment did not introduce any bug, that CompCert's semantics and printer of Assembly code did not either, and that the interfacing between Coq and the compiled program went without any hitch. The latter point is the only new potential source of inconsistency and it should be easy to make it as sound as other parts of Coq. Finally, since the language has been proved to be safe, one does not even need to prove the correctness of a program beforehand; it can be run inside Coq (*e.g.*, for experimentation purpose) without any fear that it might jeopardize the consistency of Coq by corrupting its memory.

REFERENCES

Andrew W. Appel. 2011. Verified Software Toolchain. In *20th European Symposium on Programming* (Saarbrücken, Germany) *(Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (mar 2014), 35 pages. https://doi.org/10.1145/2579080

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. 2023. Polymorphic Types with Polynomial Sizes. In *9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Orlando, FL, USA). 36–49. https://doi.org/10.1145/3589246.3595372

Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. https://doi.org/10.1109/SecDev.2018.00015

[4]Some more work might be needed if we want to accommodate ABIs that mandate that small structures, *e.g.*, complex numbers, are passed through registers.

Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 59–88. https://doi.org/10.1145/3622799

Troels Henriksen and Martin Elsman. 2021. Towards size-dependent types for array programming. In *7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. New York, NY, USA, 1–14. https://doi.org/10.1145/3460944.3464310

Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. 2020. Verification of Programs with Pointers in SPARK. In *22nd International Conference on Formal Engineering Methods* (Singapore) *(Lecture Notes in Computer Science, Vol. 12531)*, Shang-Wei Lin, Zhe Hou, and Brendan Mahony (Eds.). 55–72. https://doi.org/10.1007/978-3-030-63406-3_4

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 41 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *41st ACM SIGPLAN Symposium on Principles of Programming Languages* (San Diego, CA, USA), Peter Sewell (Ed.). 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009a. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Xavier Leroy. 2009b. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Liyi Li, Yiyun Liu, Deena Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. 2022. A formal model of Checked C. In *35th IEEE Symposium on Computer Security Foundations*. 49–63. https://doi.org/10.1109/CSF54842.2022.9919657

Guillaume Melquiond and Raphaël Rieu-Helft. 2023. WhyMP, a Formally Verified Arbitrary-Precision Integer Library. *Journal of Symbolic Computation* 115 (2023), 74–95. https://doi.org/10.1016/j.jsc.2022.07.007

Magnus O. Myreen. 2021. The CakeML Project's Quest for Ever Stronger Correctness Theorems. In *12th International Conference on Interactive Theorem Proving* (Rome, Italy) *(Leibniz International Proceedings in Informatics, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Article 1, 10 pages. https://doi.org/10.4230/LIPIcs.ITP.2021.1

Raphaël Rieu-Helft. 2020. *Development and verification of arbitrary-precision integer arithmetic libraries.* Theses. Université Paris-Saclay. https://theses.hal.science/tel-03032942

Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *8th International Conference on Principles of Security and Trust* (Prague, Czech Republic) *(Lecture Notes in Computer Science, Vol. 11426)*, Flemming Nielson and David Sands (Eds.). 76–98. https://doi.org/10.1007/978-3-030-17138-4_4

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. *ACM SIGPLAN Notices* 47, 1 (Jan. 2012), 427–440. https://doi.org/10.1145/2103621.2103709