



Towards Verifiable System Code using a DSL Compiled to Efficient and Readable C Code

Clément Chavanon

Inria, Univ Rennes, CNRS, IRISA
Rennes, France
clement.chavanon@inria.fr

Henrik Karlsson

KTH Royal Institute of Technology
Stockholm, Sweden
henrik10@kth.se

Frédéric Besson

Inria, Univ Rennes, CNRS, IRISA
Rennes, France
frederic.besson@inria.fr

Sandrine Blazy

Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Roberto Guanciale

KTH Royal Institute of Technology
Stockholm, Sweden
robertog@kth.se

Abstract

Critical embedded systems deserve the highest level of assurance to guarantee that their implementation satisfies their specification. However, there is often a gap between the verification effort carried over high-level abstractions and the source code. To tackle this problem, we propose a verification-aware DSL designed to program and prove critical embedded software. Our DSL, named Barocq, is minimal and purely functional. It makes programs amenable to verification inside the Rocq proof-assistant. Barocq comes with a compiler to C that generates efficient and human-readable code. Barocq bridges the gap between the program that is reasoned about and the program that is compiled. A key feature of Barocq is that its semantics and data-representation are zero-cost abstractions at runtime. This is achieved by enforcing, using static analysis, a strict alias-control programming discipline which enables to compile functional updates as imperative in-place mutations. Finally, we demonstrate that Barocq has adequate features to write an interesting class of critical embedded code, including most components of the S3K microkernel, which can be used as drop-in replacement for the original C code.

CCS Concepts: • Software and its engineering → Domain specific languages; Compilers; • Computer systems organization → Embedded systems.

Keywords: Domain Specific Languages, Rocq, Compilers, Microkernels

ACM Reference Format:

Clément Chavanon, Henrik Karlsson, Frédéric Besson, Sandrine Blazy, and Roberto Guanciale. 2026. Towards Verifiable System Code using a DSL Compiled to Efficient and Readable C Code. In

Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3814943.3816170>

1 Introduction

Deductive verification is commonly used to develop critical embedded software, as it offers robust mathematical guarantees that an implementation satisfies its specification. Success stories include the development of trustworthy and efficient software, for instance, cryptographic libraries [21] and authentication engines [6]. They demonstrate the scalability and practicality of this verification methodology. A primary obstacle to the daily use of deductive verification by software development teams is that they often operate on an abstract model. Indeed, the verification effort relies on designing new abstractions that facilitate the reasoning and the formulation of relevant properties. Therefore, for software, there is often a verification gap between what is proved and what is run. The Verified Software Toolchain (VST) [2] approach is to prove properties of Clight programs [4] and use the CompCert [17] verified compiler. VST requires reasoning about the subtleties of the C semantics. In particular, ruling out undefined behaviors, a prerequisite to establish stronger properties, can already be a time consuming task.

Instead of directly verifying C code, an attractive idea is to reason about higher-level languages and automatically derive a running implementation. For instance, this can be done using the Rocq prover [24] which features the dependently typed functional language Gallina, an environment to state and prove theorems, a legacy extraction mechanism [18] but also a verified extraction mechanism [10] towards an intermediate language of the OCaml compiler. This approach has the advantage that the program to be verified is expressed in a high-level language and can be compiled in a trustworthy way to executable code. One drawback is that the running program requires a runtime, in particular a garbage collector. Therefore, it is not applicable for embedded systems where resources are statically allocated and memory management needs to be finely controlled.



This work is licensed under a Creative Commons Attribution 4.0 International License.

LCTES '26, Boulder, CO, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2721-4/2026/06

<https://doi.org/10.1145/3814943.3816170>

A better approach to develop trustworthy and efficient critical embedded software is to generate C code from a functional language. This approach is pioneered by Low* [22], and Cogent [19]. Low* is a DSL embedded inside F* used to develop cryptographic libraries. Cogent is a restricted functional language embedded inside Isabelle/HOL used to implement file systems. Here, embedded means that the language is defined in the proof tool, so that it becomes feasible to reason about programs directly within the theorem prover. The expressiveness of both languages is limited, so that their compiler generates C code which is free of any trusted runtime and can inter-operate with existing C code. It is worth mentioning that Cogent has a verified compiler thus reducing the TCB to its minimum.

In this paper we develop Barocq, a middle-ground between Low* and Cogent. Like Low*, we strive to generate readable C code that is directly inter-operable with existing C code. Like Cogent, our language is purely functional¹. Unlike both, Barocq is not equipped with a sophisticated type-system. Instead, we develop a static alias control analysis to enforce the semantics invariants needed by the compiler to generate idiomatic C code (*i.e.*, close to hand-written C). Barocq is written in Gallina and targets the source language of CompCert, thus paving the way to a fully verified compiler.

Specifically, we claim the following novel contributions.

DSL design: We present Barocq, a novel functional DSL, designed to write C-like code. We spell out semantic invariants which characterise valid Barocq programs that can be compiled with in-place memory updates and therefore implement the Barocq semantics without memory allocation.

Trustworthy semantics: To reason about Barocq programs, they are embedded in Rocq, where their semantics are defined (shallow embedding). We prove in Rocq the equivalence with the original semantics of Barocq (deep embedding), thus validating our semantics.

Compiler: The Barocq compiler leverages a monadic normal form [5] to generate idiomatic and efficient C code. It includes a static analysis to check the semantic invariants and reject invalid Barocq programs.

Experiments with S3K: To validate that our setting is realistic, we have an efficient implementation in Barocq of the S3K microkernel [13] and compare it with the original C code.

Our long-term objective is to develop a proof infrastructure for end-to-end verification of system code from specifications written in Rocq to executable code. Though there is work in progress, formally proving the correctness of the compilation from Barocq to C is not a contribution of the paper. The present paper is centered around the design of the Barocq language and its compiler and how this allows generating efficient and idiomatic C code.

The rest of the paper is organised as follows. Section 2 gives an overview of Barocq and its compiler. Section 3 details the compiler frontend and how Bowman’s B-normal form helps avoiding code duplication. Section 4 presents the static analysis leveraged by the compiler to guarantee semantic invariants and perform in-place updates. Section 5 shows how to generate idiomatic C code that takes into account the memory layout. Section 6 presents our experiments with a Barocq port of the S3K microkernel. Section 7 discusses related work and Section 8 concludes.

2 From Barocq to C: An Overview

This section explains the salient features of Barocq and its compilation to idiomatic C code.

2.1 Programming in Barocq

Barocq is a minimal, first-order, simply typed, purely functional language. Barocq features primitive types (boolean, machine integers, enumeration types), arrays and records. The comprehensive syntax is given in Appendix A. For illustrative purposes, we consider a register update function of an idealised microkernel. It relies on two record types: `proc` represents processes and `kstate` represents the kernel state. A process is defined by its `pid` and its registers `r0` and `r1`. The kernel state is composed of the `pid` of the currently running process `curr_pid` and the table of processes `ptable`. We write $[\tau]$ for an array whose elements are of type τ . The registers are given by the enumerated type `reg`.

```
record proc { pid: u64, r0: u64, r1: u64 }
record kstate { curr_pid: u64, ptable: [proc] }
enum reg { R0, R1 }
```

Compiling those types using the Barocq compiler, we obtain the following C declarations. Barocq maps records to C **structs**. By default, any non-primitive value is *boxed*, *i.e.*, it is wrapped into a pointer. Thus, field `ptable` is a pointer to an array of `proc` pointers. For conciseness, we write `uint64_t` for the C type of 64-bit unsigned integers.

```
struct proc { uint64_t pid; uint64_t r0; uint64_t r1 };
struct kstate { uint64_t curr_pid; struct proc **ptable };
enum reg { R0, R1 };
```

Next, we define the function `reg_write` that puts value `v` in register `r` of the process `pid`. It first loads the process table in the local variable `procs`. The local variable `procs1` is the updated process table. It is obtained by first getting the process of interest `p` and updating its register `r` with `v`, which gives `p1`, and then by replacing `p` by `p1` in `procs[pid]`. Last, the function returns a kernel state obtained by updating the process table with `procs1`. To complete our Barocq program, we finally define a global variable `kern` of type `kstate`.

The compilation of `reg_write` is presented in Listing 2. The Barocq compiler strives to generate C code that is as readable and as idiomatic as possible. Names are normalised but the original names can be recovered by erasing the added

¹In Low*, the memory is encoded using an explicit state monad.

```

1  defn reg_write(ks: kstate, pid: u64, r: reg, v: u64): kstate =
2  let procs = ks.ptable in
3  let procs1 =
4    let p = procs[pid] in
5    let p1 = match r with R0 => p.r0<-v | R1 => p.r1<-v end
6    in procs[pid]<-p1
7  in ks.ptable<-procs1
8  @unique defn kern : kstate = { (* kernel init *) }

```

Listing 1. Example of a Barocq function for register update.

```

void reg_write(uint64_t p_pid, enum reg p_r, uint64_t p_v) {
  struct proc **u1_procs; struct proc *u1_p;
  u1_procs = (*kern).ptable;
  u1_p = u1_procs[p_pid];
  switch (p_r) { case R0: (*u1_p).r0 = p_v; break;
                case R1: (*u1_p).r1 = p_v; break;
  } return; }

```

Listing 2. C code for reg_write.

prefix. A salient property of the generated code is that all data-structure modifications are made in place as in efficient C. This is the case for the register updates $(*u1_p).r0 = p_v$ and $(*u1_p).r1 = p_v$. Moreover, Barocq intermediate variables are not present in the generated C code. Indeed, the optimisations performed in the backend exploit the semantic invariants of Barocq to detect and remove code that is dead for the imperative semantics. The @unique annotation of the global variable kern, triggers a *globalisation* optimisation [23]. This optimisation allows replacing function parameters by a global variable. In this example, kern is used in place of the ks argument.

2.2 Control over the Memory Layout

For embedded systems, the memory layout of ptable may be unsatisfactory because it introduces pointer indirections. Barocq provides a fine-grained control over the memory layout and supports selective *unboxing* of values using the operator #. Unboxing values means flattening their layout in their parent data structure. We can first unbox the processes inside the process table by associating ptable with type [#proc], which results in C type **struct** proc_t * instead of **struct** proc_t **. It is possible to go further by also unboxing the process table in the kernel state by writing [#proc; 10ul]. To unbox an array, it is necessary to specify its length (in this example, 10ul) so that the size of the parent C type is known. This results in associating ptable with C type **struct** proc ptable [10]. From the Barocq standpoint, boxed and unboxed versions of ptable are accessed the same way: it does not affect the source code of reg_write. This is the role of the Barocq compiler to adjust for the memory layout (see Section 5.2).

2.3 Enforcing Semantic Invariants by Static Analysis

Updating a record or an array in a functional style requires allocating a new object for each update. This is infeasible for embedded software, where both memory and execution time are at a premium. For this reason, the Barocq compiler generates in-place mutations for record and array updates. However, performing in-place mutations cannot be done naively as the correspondence between Barocq’s functional and C’s imperative semantics may be lost. Thus, we develop a novel static analysis that enforces semantic invariants which ensures that the Barocq compiler only accepts programs where in-place updates are guaranteed to preserve the functional model.

Let us now explain the reasoning of the Barocq’s static analysis, allowing the Barocq compiler to validate programs such as reg_write. This static analysis relies on two important properties of Barocq programs: (i) The memory is tree-shaped (there is no sharing at all); (ii) Function arguments are not aliased. The compiler tracks precisely the aliasing of local variables and for which part of the data the functional copy semantics and imperative in-place semantics coincide. To begin with, the function accesses the *i*th process of the kernel ptable. Therefore, we have two aliases $procs = ks.ptable$ and $p = ks.ptable[pid]$. When we update the register $r0$ or $r1$, we modify in-place the process p . As a result, the compiler invalidates the three paths that differ in the functional and imperative semantics, namely $ks.ptable[pid].rx$, $procs[pid].rx$ and $p.rx$ where $x \in \{0, 1\}$. If a variable containing what we call *invalid* paths is used as the right-hand-side of an update, the program is rejected. In our case, there is no such problem, as, starting from line 8, p is dead and therefore its invalid paths are harmless. Actually, the other assignments overwrite invalid paths with values that are valid and therefore restore the functional semantics. Indeed, $procs[pid]<-p1$ overwrites the invalid paths of the ptable and therefore the new ptable, *i.e.* proc1 is entirely valid. Eventually, the last assignment overwrites the invalid ptable with the valid proc1 variable. In the end, we get that the compiled reg_write function exhibits a purely functional interface (*i.e.*, the return value is completely valid), with the caveat that it invalidates the path $ks.ptable[pid]$ of the kernel argument. This is not an issue because the previous kernel state will most likely not be reused after a call to reg_write.

2.4 Overview of Compiler Architecture

The Barocq compiler (see Figure 1) performs successive passes using four intermediate representations (IR). It targets Csyntax, the source language of the CompCert C compiler [16]. Moreover, we embed Barocq in Rocq and prove that it corresponds to BarocqBNF, the first IR of the compiler based on the B-normal form described by Bowman [5]. This is essential to have a formal statement that the code that is

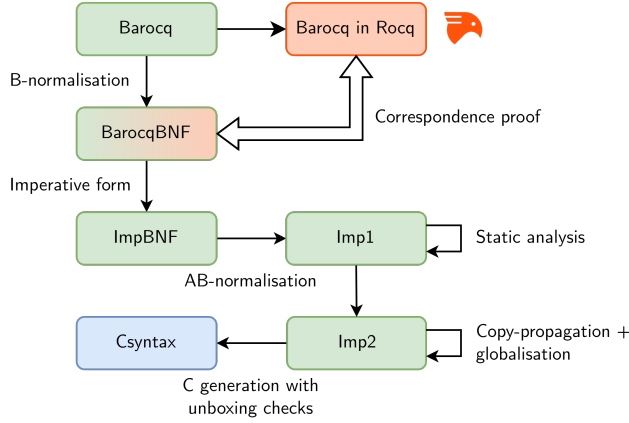


Figure 1. Barocq compilation toolchain.

reasoned about in the proof assistant is exactly the code that is compiled by the compiler.

2.4.1 From Barocq to Csyntax. Imp1 is a pivotal language because it is both associated with a copy semantics and an in-place semantics. The static analysis operates on this IR and is responsible for ensuring that the compiler only accepts programs for which both semantics exhibit the same behavior. From Barocq to Imp1, the compiler passes follow the approach of Bowman [5] and exploit the B-normal form of functional programs which, using well-conceived normalisation and IRs, turns functional code into imperative code. Another benefit is that it avoids code duplication, thus making the generated code more readable.

Imp2 is an imperative language where some standard optimisations (e.g., copy propagation) are performed to improve code readability. This is also where we take into account the memory layout: memory accesses are compiled differently depending on whether data structures are boxed or unboxed. This is also the place where we perform globalisation and replace unique function arguments by global variables.

2.4.2 Rocq Embedding and Correspondence Proof. Barocq also compiles to Gallina. However, we do not simply generate Gallina programs, we also generate a proof of correspondence between a Gallina program and its abstract syntax tree in BarocqBNF. In technical terms, we generate a *shallow* embedding of the Barocq program *i.e.*, a genuine Gallina program. Though Barocq is functional, the translation is not direct because certain Barocq constructions, e.g., array accesses or some arithmetic operations, are implicitly partial and therefore require the parsimonious insertion of an error monad. The error monad is used to propagate potential errors raised by those operations (e.g., out-of-bound array accesses and divisions by zero). The Rocq embedding of Listing 1 is shown below where `let* x := e1 in e2` is the notation for the bind operator of the error monad and `r <| fd:=v |>` is the notation for record update.

$$\begin{aligned}
 \text{Atom } \ni a &::= i \mid C \mid x \mid \diamond a \mid a_1 \bowtie a_2 \mid a_1[a_2] \mid a_1.f \\
 &\quad \mid f(a_1, \dots, a_n) \\
 \text{Comp } \ni c &::= a \mid a_1[a_2] \leftarrow a_3 \mid a_1.f \leftarrow a_2 \\
 &\quad \mid g(a_1, \dots, a_n) \\
 \text{Expr } \ni e &::= c \mid \text{match } a \text{ with } \overline{C} \Rightarrow e \text{ end} \\
 &\quad \mid \text{if } a \text{ then } e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2
 \end{aligned}$$

Figure 2. Syntax of BarocqBNF.

```

Definition reg_write (ks:kstate) (i:int64) (r:reg) (v:int64)
  : option kstate :=
  let procs := ks.(ptable) in
  let* procs1 :=
    let* p := procs.[pid] in
    let p1 := match r with
      | R0 => p <| r0 := v |>
      | R1 => p <| r1 := v |> end
    in procs.[pid←p1]
  in ret (ks <| ptable := procs1 |>).

```

Although the Gallina *shallow* embedding is visually close to the Barocq source code, we do not trust our translation. We also generate a BarocqBNF AST *i.e.* a *deep* embedding, that is equipped with a semantics and then prove that both coincide. This approach allows to establish a formal correspondence between the Gallina code and the BarocqBNF AST that is compiled to C.

3 Barocq Frontend

The frontend takes as input the Barocq source language and generates Imp1 code where nested let-bindings are turned into imperative assignments but the memory model is still functional (no in-place updates).

3.1 B-Normal Form for Barocq

Bowman [5] makes the case for a monadic B-normal form (BNF) that allows nested let-bindings. For Barocq, a crucial advantage of BNF is that this normalisation introduces neither code duplication nor nested anonymous functions.

Barocq BNF features *atoms*, *computations* and *expressions* (see Figure 2). Atoms are built from constants, enum constructors, variables, unary and binary operations, record and array accesses and side-effect free function calls. Atoms have the property that they never modify the memory. Computations include atoms, record and array updates and effectful function calls. Expressions are built from computations, conditionals, a pattern matching construct and let-bindings.

Example 3.1. The Barocq code of Listing 1 is already in BNF: `ks.ptable` and `procs[pid]` are atoms; `p.r0←v` and `p.r1←v` are computations. The rest of the expression is built from nested let-bindings which bind expressions which are either atoms, or the pattern-matching over the register `r` for which each branch returns a computation.

Because Barocq targets readable C code, the quality of the BNF matters. Intuitively, a BNF normalisation is better if it reduces the number and the nesting of introduced let-bindings. We propose a BNF normalisation which improves the quality of the BNF compared to the existing, direct, syntax-directed algorithm proposed by Bowman. The main insight of our algorithm is, unlike typical compilers, to keep untouched expressions that are already in BNF and refrain from generating nested let-bindings. Example 3.2 illustrates the benefit of our optimised algorithm.

Example 3.2. Consider the expression where the functions f, g and h are effectful: $f(g(h(x, y), z), w).fd \leftarrow 20$. The BNF obtained by Bowman’s algorithm (Figure 3a) has 4 let-bindings and a nesting level of 3. With our algorithm (Figure 3b), we get a BNF with 3 let-bindings and no nesting.

<pre> let b1 = let b2 = let b3 = h(x, y) in g(b3, z) in f(b2, w) in let b4 = 20 in b1.fd ← b4 </pre> <p>(a) Bowman’s BNF</p>	<pre> let b1 = h(x, y) in let b2 = g(b1, z) in let b3 = f(b2, w) in b3.fd ← 20 </pre> <p>(b) Barocq’s BNF</p>
--	---

Figure 3. B-normalised forms for $f(g(h(x, y), z), w) * 20$.

3.2 From BNF to Imp1

The next two passes from BarocqBNF to ImpBNF and ImpBNF to Imp1 follow directly from Bowman. The main insight of the first pass is that lexically scoped let-bindings are compiled into nested, imperative, globally-scoped assignments to local variables. Imp1 has a classic imperative syntax that is given in Figure 4. We have a `ret` statement which will always be the last executed instruction. We also have a conditional and a pattern-match statement. In Imp1, there is no nested assignment; the right-hand side c of $x := c$ must be a computation. In order to generate Imp1 code, it is necessary to reorder the assignments so that inner assignments are performed before outer assignments. It is also necessary to propagate assignments within conditionals and the branches of pattern-matchings.

Example 3.3. Translation of Listing 1 into Imp1 code

```

1  u1_procs = p_ks.ptable;
2  u1_p := u1_procs[i];
3  match p_r with
4    R0 ⇒ u1_p1 := u1_p.r0 ← p_v
5    | R1 ⇒ u1_p1 := u1_p.r1 ← p_v end;
6  u1_procs1 := u1_procs[i] ← u1_p1;
7  res := p_ks.ptable ← u1_procs1;
8  res

```

The inner assignment to `u1_p` is now syntactically performed before the assignment of `u1_procs1`. The assignment to `u1_p1`

$$\begin{array}{lcl}
 \text{Statement } \ni s & ::= & \text{ret } a \mid x := c \mid s_1; s_2 \\
 & & \mid \text{if } a \text{ then } s_1 \text{ else } s_2 \\
 & & \mid \text{match } a \text{ with } \overline{C} \Rightarrow s \text{ end}
 \end{array}$$

Figure 4. Syntax of Imp1.

is duplicated in both branches of the pattern-matching. Eventually, `u1_procs1` and the result `res` are updated.

Imp1 has global scopes and an imperative control-flow. Yet record and array updates are still functional. In the next section, we present our static analysis based approach that verifies whether in-place updates can be performed.

4 Static Analyses for In-Place Updates

In order to compile functional updates into imperative in-place ones, we need to show that the functional values will not be used after the update has been performed. Linear typing disciplines achieve this goal by simply forbidding a value to be used twice. Our approach consists in enforcing strict alias control using a static analysis of our own. Intuitively, it tracks down which part of the functional values cannot be used anymore because they have been updated in-place. Therefore, we decompose a value into access paths and compute an over-approximation of the paths that are *invalid* i.e., the values we obtained by following the paths that may not coincide in the functional and imperative interpretation of the code. To avoid an over-pessimistic approximation, the analysis of invalid paths is a must-alias analysis, and is able to collect all the paths that are (in-)directly impacted by a memory update. Our static analysis is modular, i.e., it analyses each function independently. In full generality, devising modular alias analyses may be a very hard problem. However, our fairly simple analysis is powerful enough to enforce the strict Barocq programming discipline.

Moreover, the analysis exploits the three key invariants of Barocq programs, namely the memory is tree-shaped, only primitive values can be mutated, and function arguments are not aliased. These invariants are enforced by the must-alias analysis. It maintains a must-points-to graph where nodes represent memory locations and edges are labelled by access fields.

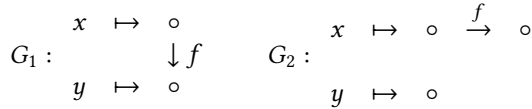
4.1 Must-Alias Analysis

A pair of access paths $(x.f_1 \cdots f_n, y.f_1 \cdots f_m)$ is in a must-alias relation if, in the must-points-to graph, there are paths $x \mapsto n_0 \xrightarrow{f_1} \cdots \xrightarrow{f_n} n_n$ and $y \mapsto m_0 \xrightarrow{f_1} \cdots \xrightarrow{f_m} n_m$ such that $n_n = n_m$, namely the paths lead to the exact same node.

At function entry, each non-primitive argument is mapped to a distinct node. Providing y is not a primitive value, an assignment $y := x.f_1 \cdots f_n$ augments the graph with a path $x \mapsto n_0 \xrightarrow{f_1} \cdots \xrightarrow{f_n} n_n$ and binds the variable y to n_n . In

general, dealing with memory updates is more challenging because there are must-alias paths (those in may-alias with the update) that need to be invalidated. Here, we exploit the fact that *only primitive values may be mutated*. In other words, throughout the program, aliases are invariant. The consequence is that for any side-effect either the must-alias graph is unchanged or the program is rejected.

Example 4.1. Consider the assignment $x_1 := (x.f \leftarrow y)$ and the must-alias graphs G_1 and G_2 .

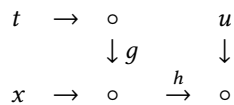


In G_1 , $x.f$ and y are already in must-alias. As a result, the side-effect has no effect, this is a no-op and the graph G_1 is still a correct approximation of alias information after the assignment. We only need to add an edge from x_1 to the destination node of x . The situation of G_2 differs. As $x.f$ points-to a memory address distinct from y , the side-effect would mutate the memory. This violates our invariant and therefore the program is rejected.

4.2 Invalid Path Analysis

The invalid path analysis is using the alias analysis to invalidate paths in memory where the functional semantics and the imperative semantics differ. In theory, when an update is performed, all the paths that are in a *may* alias relation need to be invalidated. However, our alias analysis is only performing a *must* alias analysis. Actually, because we rule out any side-effects that modify the memory layout, the may-alias information can be entirely rebuilt from the must-alias graph. We are in a sweet spot where either we have exact must-alias information or no alias information at all.

Example 4.2. Consider the graph G_3 and again an assignment to field f ($x_1 := (x.f \leftarrow y)$).



Suppose that y holds a primitive value. Then, the assignment does not create must alias information. However, the memory is updated and therefore paths need to be invalidated. The value of field $x.f$ is updated. So, its functional value cannot be used anymore and we invalidate the path $x.f$. The update has no impact on parts of the memory that are accessible from other un-aliased fields. For instance, the invalid paths of u (if any) are unchanged. However, any path p that is in may alias relation with x , *i.e.*, $t.g$, has its field f invalidated.

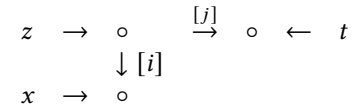
The invalid path analysis processes one function at a time, starting from the leaves of the call graph, and computes an invalid path summary for each function. We mandate the return value to be free of invalid paths. However, for the

arguments, we record their invalid paths that are used when we model the effect of function calls.

4.3 Extension to Arrays

The presentation given above works for records. To accommodate arrays, adjustments need to be made. Conceptually, arrays are records with dynamic field names. For the must-alias graph, edges may be either record fields or array access expressions. A difference is that, contrary to record fields, array access expressions are mutable. As a result, as soon as a variable is modified, all the array expressions mentioning it are invalidated *i.e.*, in the graph, they are replaced by a $\top$ value. The other difference is that record fields that are syntactically distinct are never aliased. For array accesses, this is not the case. Therefore, we make the worst-case assumption and assume that they may alias.

Example 4.3. Consider G_4 where an array x has must-aliases to i and j ; and the record update $x_1 := (x.f \leftarrow y)$.



Similarly to the record case, $z.[i].f$ needs to be invalidated. Moreover, because i and j may have the same value, the path $z.[j].f$ also needs to be invalidated. Another indirect consequence is that $t.f$ is also invalidated. In conclusion, assuming that array access may alias, we give a may-alias interpretation to the point-to graph obtained by recursively merging nodes that are reached from an array expression that could have the same value at runtime.

5 Barocq Backend and Generation of C

The first pass of the backend runs the must-alias and invalid path analyses presented above to check that updates can be performed in-place. To comply with the constraints of the Imp2 language, it also splits a side-effect assignment into a side-effect instruction and a pure assignment, namely

$$x := r.f \leftarrow v \rightsquigarrow r.f \leftarrow v ; x := r$$

For the class of programs that are successfully analysed, all updates to non-primitive values are no-op because the rhs and lhs are necessary in must-alias. Therefore, the translation to Imp2 optimises them away. To propagate the introduced copies but also eliminate copies which are actually dead code, we perform a standard copy propagation pass.

Example 5.1. For our running example, the Imp1 code (see Figure 3.3), we get the following Imp2 code.

```

1  u1_procs = p_ks.phtable; u1_p := u1_procs[i];
2  (match p_r with R0 => u1_p.r0 <- p_v | R1 => u1_p.r1 <- p_v end);
3  ret p_ks

```

Within the pattern-matching, the assignments are first split into a side-effect followed by an assignment $u1_p1 := u1_p$.

$$\begin{aligned}
G[a_1[a_2]] &= \begin{cases} G[a_1][G[a_2]] & \text{if } a_1[a_2] \text{ is boxed} \\ G[a_1] + G[a_2] & \text{if } a_1[a_2] \text{ is unboxed} \end{cases} \\
G[a_1[a_2] \leftarrow a_3] &= G[a_1][G[a_2]] = G[a_3] \quad \text{if } a_3 \text{ is primitive}
\end{aligned}$$

Figure 5. Imp2-to-Csyntax translation function.

Using copy propagation, the rest of the program is renamed using the mapping $u1_p1 \mapsto u1_p$. However, because the assignment $u1_procs[i] \leftarrow u1_p1$ is a no-op, it is optimised away. Hence, the variable $u1_p1$ is dead and the assignment is simply eliminated.

5.1 Globalisation

Globalisation [23] is a compiler optimisation which consists in replacing function arguments by global variables. In general, the advantage is that this reduces the cost of function calls. For embedded systems, a common pattern consists in statically allocating global variables that are manipulated by the program. Instead of passing global variables as arguments of most functions, it is quite natural to directly access and update variables within the function body. In Barocq, we generate this typical C idiom guided by the `@unique` annotation which triggers globalisation of the following variable declaration. More precisely, a declaration of the form `@unique defn var:  $\tau$  =  $e$` means that `var` is the only instance of type τ . Based on this information, the Barocq compiler checks that any argument of type τ can be globalised by the variable `var`. This includes checking that there is a single argument of type τ but also that τ is a top-level type that is not used as a field record or an array element.

Example 5.2. For our running example, variable `kern: kstate` is declared `@unique`. So, after globalisation, we get the signature `reg_write(pid: u64, r: reg, v: u64): void`. As expected, the globalisation of the variable has the effect of erasing the first argument. Also, instead of returning an updated kernel, after globalisation, the function has the return type `void`.

5.2 Layout-Aware Generation of C

The last pass generates genuine C code. There is almost a one-to-one mapping between Imp2 and C instructions with the caveat that the data layout needs to be taken into account. As explained in Section 2.1, Barocq type declarations are annotated with unboxing directives. The translation function for array access and update is given in Figure 5 (the treatment of records is similar). The translation is type-directed. Given an array access of the form $a_1[a_2]$, the atom a_1 and the index a_2 are recursively translated. If the array element has a boxed type, the access is compiled into a standard C array access. Otherwise, the array element has a `#` annotation and is unboxed. In that case, the translation gets a pointer to the a_2^{th} element of the array a_1 . This is done by performing

the address computation $G[a_1] + G[a_2]$ used to compile the array access. For side-effects, the translation requires the type of the elements to be primitive types. For Barocq, this is not a restriction because we have the guarantee that side-effects with non-primitive values have been ruled out. This condition could be relaxed to include boxed types but unboxed types need to be excluded as C rejects such assignments: updating an array element with *e.g.*, a record, is not permitted. A workaround could be to perform an explicit copy. However, this would incur a (hidden) runtime cost which goes against the design principle of Barocq.

Eventually, we get a C code specialised for the memory layout. Thanks to Barocq, a single annotation in the type declaration allows to change the memory layout of the whole program. Moreover, the semantic invariants ensure that this can always be done without any hidden runtime cost.

6 S3K Case Study

We evaluate the expressiveness and performance of Barocq using as a case study the S3K partitioning kernel [13]. S3K is a capability-based microkernel designed for safety- and security-critical RISC-V embedded systems (see Section 6.1). The original S3K implementation² is reasonably small (2,000 lines of C/Assembly) yet representative. It exercises core language features, including arrays, records, and foreign function calls, while requiring efficient code generation to satisfy real-time constraints. Section 6.2 explains how we implemented the vast majority of the kernel’s core logic in Barocq. A small amount of easily inspectable C/Assembly glue code remains to handle loops and hardware interactions (see Section 6.2.3). Section 6.4 presents the evaluation results.

6.1 A Primer on S3K

S3K supports a fixed set of processes that own and manage resources using capabilities. All kernel data-structures are statically allocated during kernel startup. The kernel therefore operates with fixed upper bounds on the number of processes and capabilities. This design avoids dynamic memory allocation and allows most kernel structures to be implemented as static arrays, which aligns well with the programming model of Barocq.

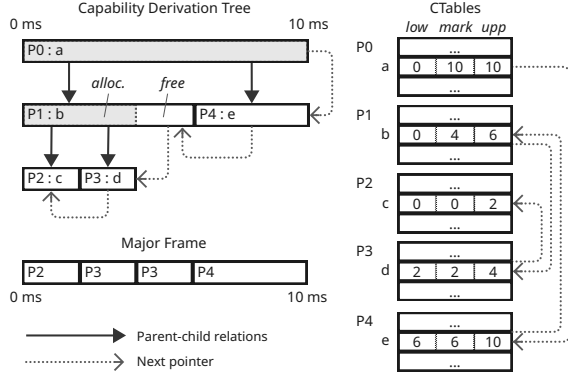
There are seven types of capabilities in S3K: **Time slice** for managing intervals of execution time; **Memory slice** for managing physical memory; **PMP frame** for managing the RISC-V’s PMP memory protection; **Monitor slice** for managing other processes; **Channel slice** for managing IPC channels; **Server and Client sockets** for communicating.

Each capability is encoded using a 64-bit tagged union (see Figure 6a). Field type gives the capability type. Slice

²The RTSS’25 paper showcases S3Kv2, a new version of the kernel that was not yet complete when the port of S3K to Barocq started. The choice of implementing S3Kv1 was also motivated by the fact that it is a more demanding case study for Barocq, as it contains complex data structures such as linked lists that S3Kv2 got rid of.

	type (4 b)		data (60 b)		
time	1	-	low	mark	upp
mem	2	rwX	low	mark	upp
pmp	3	rwX	slot	addr. enc.	
	...				

(a) Capability data.



(b) Capability derivation tree (upper-left) for time capabilities, its ctble representation (right), and corresponding minor-frames (below).

Figure 6. S3K capabilities.

capabilities control a *range* of resources, identified by low and upp fields. The interval from low to mark represents resources that have already been delegated to children, while interval from mark to upp represents resources free to use. Some capabilities contain an additional field, for instance the field *rwX* of memory slice capability describes the associated read, write and execute permissions.

Following seL4 [14] and Barrelfish [3], S3K implements a capability derivation tree (CDT): a tree structure where a capability is an ancestor of another if its range includes the range of the other capability. The CDT is implemented as a doubly linked list where children directly follow their parents, as illustrated in the upper left-hand side of Figure 6b. This structure is distributed among fixed-size per-process capability tables (ctble). Each element of this table consists of an 8-byte capability and two 8-byte indices, next and prev, for the doubly-linked list.

Processes use and manage the capabilities using system calls, referencing the corresponding capability by its ctble index. There are three groups of such system calls: read, move and delete allow a process to manage its capabilities table. Derivation enables the creation of new capabilities from existing ones. To prevent privilege escalation, the newly derived capabilities grant access to a subset of the resources provided by the original. Newly derived slice capabilities take their resources from the free segment of the parent capability. Once derived, the revocation allows for the reclamation of resources by deleting derived capabilities. Capability revocation recursively deletes children in a right-to-left, top-down

order, increasing the free segment whenever possible. To bound WCET of system calls, capability revocation can be interrupted, potentially freeing only a subset of children and their associated resources. A successful revocation frees the entire span of the slice.

6.1.1 Memory Management. S3K leverages RISC-V Physical Memory Protection (PMP) to isolate processes using two capability types: memory slices and PMP frames. While a memory slice grants ownership and management of a contiguous physical memory region, a derived PMP frame serves as the mechanism to configure the hardware and enable actual memory access.

6.1.2 Inter-Process Communication and Monitoring. S3K provides two main inter-process interaction mechanisms. The channel capabilities allows establishing but also transferring client/server socket capabilities among processes. These sockets can be used to send/receive messages, providing a mechanism to grant access permissions to other processes. The monitoring capability allows a process to manage other processes. A process with a monitor capability exercises complete control over the state of the monitored processes. This control extends to actions such as suspending, resuming, transferring capabilities, and modifying registers.

6.1.3 Time Management. S3K introduces a novel time-driven scheduler that combines traditional static scheduling with dynamic capability-based time allocation. Execution across hardware threads is partitioned into fixed, repeating major frames, which are further divided into minor frames. These minor frames are managed via time slice capabilities represented by the free segments in Figure 6b. By manipulating these capabilities, a process can dynamically allocate or reclaim minor frames for its sub-processes. Crucially, S3K's design ensures that these reconfigurations never affect the minor frames of unrelated processes.

6.2 Barocq Port of the S3K Kernel

Porting S3K to Barocq was a step-by-step process. The newly Barocq-implemented parts were incrementally included in the existing C code base. Re-writing S3K also helped improving Barocq by adding more features to the language, in particular a file-based module system, abstract type, constant and function declarations, as well as unboxed values. Those features are essential to make Barocq a practical and suitable language for microkernel development. The two S3K implementations are comparable in terms of lines of code. S3K-C is about 2000 lines of C (.c only) and 200 lines of Assembly, whereas S3K-Barocq is about 2000 lines of Barocq, about 350 lines of C (.c only) and 200 lines of Assembly.

6.2.1 Data Representation. We encode capabilities using plain unsigned 64-bit integers and implement the corresponding getters and setters. For example, the getter and setter of field *low* of memory capabilities are given below.

```

defn memory_get_low(cap: u64) : u64 = (cap >> 16ul) & 0xFFFFul
defn memory_set_low(cap: u64, v: u64) : u64 =
  (cap & ~0xFFFF0000ul) | (v << 16ul)

```

The kernel state is defined as follows. The first four fields are the same main tables as in S3K-C. Field `mc` explicitly exhibits the hardware state and is discussed in more detail in Section 6.2.2. Field `message` is used to store the IPC messages sent from the user space. The last three fields are used when a Barocq function needs to return several results, e.g. a kernel state plus an error code or the pid of the next process to run.

```

record kstate {
  ctable : #[cte; s3k_proc_cnt * s3k_cap_cnt],
  tslots : #[u64; s3k_slot_cnt],
  ptable : #[#proc_t; s3k_slot_cnt],
  channels : #[#channel; s3k_chan_cnt],
  mc : machine, msg : #message, vregs : #[u64; 8UL],
  errcode : err_t, next_pid : pid_t }

```

6.2.2 Hardware Interactions. S3K-Barocq relies on a Foreign Function Interface (FFI) to interact with the hardware. The FFI operates on the abstract type machine and several abstract functions that read hardware registers. By *abstract* we mean definitions that do not have an implementation in Barocq, but do have one in C or Assembly. Here is an excerpt of the hardware FFI defined in S3K-Barocq.

```

type machine of "struct"
decl timeout_read : (@read machine, u64) -> u64

```

We tell the compiler that machine is implemented as a C struct. Then, we declare the function `timeout_read` that reads the time compare register of a given hardware thread. The `@read` annotation informs the static analysis that the first argument remains completely valid after a call to `timeout_read` as this function does not mutate the machine state. The opposite annotation `@write` (not shown here) tells the static analysis that an abstract function modifies the corresponding argument and returns its updated version. This annotation prevents post-call use of this argument by invalidating it.

6.2.3 C Parts of S3K-Barocq. S3K-Barocq is a mixed implementation between Barocq, C and Assembly. The remaining C code can be divided in 3 categories. (i) C code that interacts with the hardware contains inline Assembly to read and write hardware registers. (ii) C code that contains bounded loops. This concerns the kernel initialisation, two scheduling functions and the outer loop of the capability revocation function. (iii) C code that contains unbounded loops. This is the case for only one S3K function which is the trap handler. Indeed, as the kernel may not find a process to schedule next, the trap handler may loop forever. For instance, this could happen if the capabilities are misconfigured at kernel initialisation. The original C code actually had this possibly non-terminating loop in the main scheduler function. As we wanted to express as much as possible in Barocq, we re-structured the code by moving this loop

waiting for a next process to be chosen in the trap handler instead, and implemented the main scheduling function in Barocq.

6.2.4 Accessing Barocq Definitions in C. To avoid relying on custom annotations for C programs to access Barocq definitions, we made the interface of the Barocq-generated C code predictable and very close to the Barocq source program. First, all global identifiers (types, constants, functions and enum constructors) are preserved and just prefixed with the name of the Barocq module in which they are defined. Secondly, every Barocq function typed $(\tau_1, \dots, \tau_n) \rightarrow \tau_r$ is simply compiled to a C function with signature $(\tau_1^c)(\tau_2^c, \dots, \tau_n^c)$, where τ_i^c is the C type corresponding to τ_i . An exception for that is when the annotation `@unique` is used (cf. Section 5.1).

6.2.5 Towards More C-Style Barocq Code. To get closer to the existing C version, Barocq needs support for tagged unions, bitfields and tuples. Concerning unions and bitfields, we believe this can be done without overhead for types that fit an integer, like capabilities do. To improve further the prototypes of the generated S3K functions, a solution would be to support returning a tuple of results and use a more aggressive version of globalisation to only generate C code returning a single result. For instance, a Barocq function returning both a kernel state and an error code would be compiled into a C function returning an error code and updating a global `kstate` variable. Future support of bounded loops will also help reducing the proportion of remaining C in S3K-Barocq.

6.3 Correspondence of the S3K Rocq Embeddings

To check that the translation from S3K-Barocq to S3K-Rocq is correct, we automatically generate the shallow and deep embeddings and the correspondence proof between them. Using a laptop equipped with an Intel Core Ultra 7 processors running Linux with 16 GB of RAM, the compilation of all of S3K Rocq files involved in the correspondence proofs take about 15 minutes. Note that this duration is completely unrelated from the compilation time of S3K-Barocq to C. Correspondence proofs are generated to ensure that the translation from Barocq to Rocq is correct. Thus, they will most likely be checked only infrequently, specifically when a stable version of the kernel on which verification efforts are required is released.

6.4 Evaluation of the Generated Code

We evaluate S3K-Barocq by comparing it to S3K-C. Given the importance of the IPC in microkernels, the evaluation focuses on measuring its performance.

6.4.1 Benchmark Setup. The benchmark scenario is presented in Figure 7. It involves two processes, a server and a client, that communicate with each other. The client sends a message to the server by invoking the `Call` system call

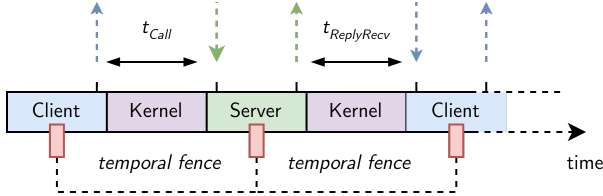
Table 1. Cost of Call, ReplyRecv and their sum RTT.

(a) Hot-cache, compiled with gcc -O2 -flto -falign-functions=16.

Time (cycles)	C	Barocq	Overhead
Call	657	690	5.0%
ReplyRecv	635	639	0.6%
RTT	1292	1329	2.9%

(b) Cold-cache, compiled with gcc -O2 -flto.

Time (cycles)	C	Barocq	Overhead
Call	2337	2608	11.6%
ReplyRecv	2410	2540	5.4%
RTT	4747	5148	7.8%


Figure 7. Benchmark scenario and measured data.

and waits for a reply. When sending the message, the client also transfers its minor frames so that the server can directly execute. When the server receives the message, it invokes ReplyRecv to send its reply and accepts the next client's message. We compute the Round-Time Trip (RTT) of the client message by summing the measured times t_{Call} and $t_{ReplyRecv}$. Those measurements are repeated 3×50 times (3 separated runs, each run taking 50 measures) and we compute the mean of those measurements. We conduct the evaluation in hot cache and cold cache settings. For the hot-cache scenario, the client and the server prefill the cache by repeating the message exchange. In the cold-cache scenario, those two processes use the temporal fence to flush the cache before executing the system calls Call and ReplyRecv (cf. Figure 7).

The benchmarks are run using a dual-core Cheshire SoC running on the Genesys 2 FPGA. The SoC features two 50 MHz CVA6 cores with 16 kB data and 16 kB instruction L1 caches. Moreover, the SoC has a shared 128 kB LLC/SRAM, and 8 GB of DRAM from the Genesys 2 FPGA.

6.4.2 Experimental Results. The kernel elf sizes are similar when compiling with gcc -O2 -flto: 17.3 kB for S3K-C and 16 kB for S3K-Barocq. The elf size produced from S3K-Barocq is lower because the Barocq compiler generates a unique C file, which is more conducive to optimising the binary size, whereas S3K-C is split into multiple source files.

The cost of Call and ReplyRecv is given for the hot-cache setting in Figure 1a and for the cold-cache setting in Figure 1b. In the former case, we use the flag -falign-functions=16 because it both increases S3K-C and S3K-Barocq IPC performances, the increase being more important for S3K-Barocq. In the hot-cache scenario, the code generated from S3K-Barocq introduces an overhead of 5% for Call, 0.6% for ReplyRecv and 2.9% for the total Round-Time Trip. In the cold-cache scenario, function alignment does not have a huge

impact on the RTT cost for Barocq, and decreases the performance of S3K-C, so we do not use the alignment flag. The overhead introduced by S3K-Barocq is 11.6% for Call, 5.4% for ReplyRecv and 7.8% for RTT. We believe that this overhead is partly induced by the difference in memory accesses. We indeed also measure the number of memory accesses during the whole communication for both of the implementations. We obtain that S3K-Barocq loads 2.2% fewer values from memory, but executes 8.3% more memory writes than S3K-C. This overhead in memory writes is due to how S3K-Barocq is written compared to S3K-C. Indeed, in S3K-Barocq, most of the functions that need to update a process or a capability access it from the global kernel state, by taking as input its identifier in the process/capability table. On the contrary, in S3K-C, the behavior of the code is more *local* in the sense that it passes the corresponding pointer to those functions, without having to re-access the value from the global variables.

7 Related Work

To bridge the gap between what is proved and what is run, many proof assistants feature program extraction mechanisms to general purpose functional languages [12, 18]. To reduce further the TCB, the extraction of Rocq is verified [10] and generates OCaml code. For HOL4, Myreen and Owen [1] propose an automatic translation validation approach between functional HOL4 programs and a fragment of SML. The CakeML verified compiler [15] also features a proof-producing HOL frontend. For a general purpose functional language, the runtime features a garbage collector, either trusted or verified [8]. In either case, this is not acceptable for embedded systems for which memory is a scarce resource and data-structures are often statically allocated.

To cover specific application domains, various DSLs have been proposed. Rupicola [20] compiles Rocq to efficient imperative code using a relational compilation approach which reduces code generation to proof-search. Because the proof-search is extensible, it can be fine-tuned on a program basis to perform aggressive optimisations that are out-of-reach of traditional compilers. Barocq follows a more traditional approach where the compiler is fixed with the advantage that the result is more predictable. Vale [11] targets cryptographic libraries and embeds in F* a subset of x64 assembly. Closer to our work is KaRaMeL [22] which compiles Low*, another

DSL embedded in F*, into C. They also share the goal of generating readable C code. The main difference with Barocq is that Low* code runs inside a state monad with an explicit C-like memory model. So, the updates of the monadic state directly compile into side-effects in C. In our design, there is no explicit state (or state monad), it is the role of the Barocq compiler and its static analysis to reject programs for which array and record updates cannot be performed in-place.

Cogent [19] is a DSL embedded in Isabelle to ease writing and proving operating system components. Its compiler is verified while the Barocq compiler is trusted. Yet we ensure a formal correspondence between the shallow and deep embeddings of Barocq. The Cogent and Barocq languages differ. Whereas Cogent is a richer functional language with higher-order functions and variant types, Barocq supports arrays that are of the utmost importance for our application domain. On the compiler side, Cogent leverages uniqueness types to perform in-place updates. Barocq is using an alias analysis which does not require user annotations and can sometimes be more flexible but less predictable. Cogent and Barocq are also using different IR. While Cogent is using a standard ANF [9], Barocq is based on the richer monadic form [5]. This avoids code duplication and reduces the amount of spurious intermediate variables, thus improving the readability of the generated C code.

8 Conclusion and Future Work

We present Barocq, a first-order and minimal functional language embedded in Rocq. By a formal correspondence between the Rocq code and the Barocq AST, we ensure the absence of representation mismatch. Our successful port of a substantial part of S3K is a positive insight that Barocq offers features tailored to the development of microkernels for embedded systems. In addition, the Barocq compiler generates readable and reasonably efficient C code that can interoperate with the existing code base. As future work, we intend to extend Barocq with bounded loops and investigate the formal verification of the key passes of the compiler.

Data-Availability Statement

An artifact for this paper is available on Zenodo [7]. The latest version of Barocq is available on [Software Heritage](#) and the S3K-Barocq development can be found on [GitHub](#).

Acknowledgments

This work was partially funded by the France 2030 project ANR-22-PECY-0005 “SecurEval”.

A Core Syntax of Barocq

The core syntax of Barocq is given below. A Barocq program is a sequence of constant and function definitions and declarations. Declarations (also called abstract definitions) are constants and functions whose implementation is provided

in C or Assembly, not in Barocq. Global constants are defined by constant expressions Cexpr. Function bodies are expressions Expr, typical of functional programming languages. The built-in operator $\leftarrow$ is used to update record fields and array elements. Barocq functions are uncurried, thus function arguments are grouped into tuples. Barocq supports the usual primitive types (booleans, integers, enums), as well as records, arrays and user-declared abstract types.

Type $\ni \tau ::=$

bool	<i>Booleans</i>
i32 u32 i64 u64	<i>Integers</i>
$\{C_1, \dots, C_n\}$	<i>Enumerations</i>
$[\tau]$	<i>Arrays</i>
$\{f_1: \tau_1, \dots, f_n: \tau_n\}$	<i>Records</i>
$(\tau_1, \dots, \tau_n) \rightarrow \tau_r$	<i>Functions</i>
$\alpha \in \text{Ident}$	<i>Abstract types</i>

Expr $\ni e ::=$

$l \in \text{Bool} \cup \text{Int}$	<i>Bool or Int literal</i>
C	<i>Enum constructor</i>
x	<i>Variable</i>
$e \text{ as } \tau$	<i>Cast</i>
$\diamond e$	<i>Unary operation</i>
$e_1 \bowtie e_2$	<i>Binary operation</i>
$e_1[e_2]$	<i>Array get</i>
$e_1[e_2] \leftarrow e_3$	<i>Array set</i>
$e.f$	<i>Record projection</i>
$e_1.f \leftarrow e_2$	<i>Record update</i>
$g(e_1, \dots, e_n)$	<i>Function call</i>
if e_1 then e_2 else e_3	<i>Conditional</i>
match e_c with $\overline{C} \Rightarrow e$ end	<i>Pattern matching</i>
let $x = e_1$ in e_2	<i>Let binding</i>

Cexpr $\ni ce ::=$

$l \in \text{Bool} \cup \text{Int}$	<i>Bool or Int literal</i>
x	<i>Constant variable</i>
$[ce_1, \dots, ce_n]$	<i>Array literal</i>
$\{f_1 = ce_1, \dots, f_n = ce_n\}$	<i>Record literal</i>
$ce \text{ as } \tau$	<i>Cast</i>
$\diamond e$	<i>Unary operation</i>
$e_1 \bowtie e_2$	<i>Binary operation</i>

Program $\ni p ::=$

$\text{defn } c: \tau = ce$	<i>Constant definition</i>
$\text{defn } g(x_1: \tau_1, \dots, x_n: \tau_n): \tau = e$	<i>Function definition</i>
$\text{decl } a: \tau$	<i>Abstract definition</i>
$p_1; p_2$	<i>Program sequence</i>

References

- [1] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. 2020. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.* 64, 7 (2020), 1287–1306. doi:10.1007/S10817-020-09559-8
- [2] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 1–17. doi:10.1007/978-3-642-19718-5_1
- [3] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 29–44. doi:10.1145/1629575.1629579
- [4] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. doi:10.1007/S10817-009-9148-3
- [5] William J. Bowman. 2024. A Low-Level Look at A-Normal Form. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 165–191. doi:10.1145/3689717
- [6] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clément Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas G. Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua M. Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 2025. Formally Verified Cloud-Scale Authorization. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2508–2521. doi:10.1109/ICSE55347.2025.00166
- [7] Clément Chavanon, Henrik Karlsson, Frédéric Besson, Sandrine Blazy, and Roberto Guanciale. 2026. *Artifact for "Towards Verifiable System Code using a DSL Compiled to Efficient and Readable C Code", at LCES'26*. doi:10.5281/zenodo.20086279
- [8] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *J. Autom. Reason.* 63, 2 (2019), 463–488. doi:10.1007/S10817-018-9487-Z
- [9] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. doi:10.1145/155090.155113
- [10] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI (2024), 52–75. doi:10.1145/3656379
- [11] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.* 3, POPL (2019), 63:1–63:30. doi:10.1145/3290376
- [12] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6009)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer, 103–117. doi:10.1007/978-3-642-12251-4_9
- [13] Henrik Karlsson and Roberto Guanciale. 2025. Partitioning Kernel With Capability Controlled Temporal and Spatial Partitioning. In *IEEE Real-Time Systems Symposium, RTSS 2025, Boston, MA, USA, December 2-5, 2025*. IEEE, 68–81. doi:10.1109/RTSS66672.2025.00015
- [14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. doi:10.1145/1629575.1629596
- [15] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. doi:10.1145/2535838.2535841
- [16] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814
- [17] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. doi:10.1007/S10817-009-9155-4
- [18] Pierre Letouzey. 2002. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, 200–219. doi:10.1007/3-540-39185-1_12
- [19] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* 31 (2021), e25. doi:10.1017/S095679682100023X
- [20] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 918–933. doi:10.1145/3519939.3523706
- [21] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. 2020. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 899–918.
- [22] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. doi:10.1145/3110261
- [23] Peter Sestoft. 1989. Replacing function parameters by global variables. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (Imperial College, London, United Kingdom) (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 39–53. doi:10.1145/99370.99374
- [24] The Rocq Development Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.15149629