Compiling Lean programs with Rocq's extraction pipeline

MPRI internship report

Simon Dima, 2025-10-01

Advised by Yannick Forster, Cambium team, Inria Paris

Summary

General context

Writing correct programs is difficult, but desirable. The highest degree of confidence in the correctness of software can be achieved by using formal methods for verification, which provide mathematical assurances about the behavior of programs. Many formal techniques, including approaches based on abstract interpretation or semiautomatic checking of verification conditions derived from Hoare triples, consider programs "from above", representing programs as formal objects and proving properties about them in some exterior formal system.

Proof assistants based on dependent type theory, such as Lean (De Moura and Ullrich 2021) and Rocq (The Rocq Development Team 2025), permit an alternative approach, in which the language used to express logical reasoning is also used directly to write the program under consideration. This enables the expressive dependent type system of the proof assistant to be used throughout the implementation of the program, expressing sophisticated constraints on the behavior of programs through the type system and allowing data to be manipulated together with proofs concerning it.

The interactive session of proof assistants often features an interpreter for the term language, but in order for programs written in a proof assistant to be executed with reasonable performance, a compilation process producing a binary executable is necessary. We focus on two such systems:

- Lean 4 and its new self-hosted compiler, which targets the C language
- Rocq's verified extraction pipeline (Forster, Sozeau, and Tabareau 2024) using the OCaml compiler.

At a high level, the Lean and Rocq pipelines both perform the task of transforming type theory into executables. However, the approaches used are quite different: the Lean compiler applies several optimization passes and progressively transforms high-level expressions into low-level C, notably handling the addition of reference counting operations as a mechanism for memory management, whereas Rocq's verified extraction pipeline only minimally alters the structure of programs, with the main step being type and proof erasure via conversion into an untyped language named $\lambda_{\square}$ ("lambda-box"), and leaves optimizations and memory management to the OCaml compiler backend and runtime.

Research problem

Implementing a full compiler, as the Lean developers chose to do, adds the correctness and performance of the compiler's optimization passes, as well as the maintenance of the associated code, to the concerns of proof assistant developers. We investigated whether implementing a simpler extraction procedure, following the path of Rocq's verified extraction to use the established OCaml compiler, would be a viable alternative for Lean. OCaml's Flambda optimizer and the runtime implementation of garbage collection are regarded as performant, mature technology used in industry, and hence we considered it plausible a priori that compiling Lean code via OCaml might provide comparable or better performance relative to the existing Lean compiler, which is newer and developed by a smaller team. A direct comparison between the two can provide useful information to guide future development of extraction from proof assistants in any case, either by working towards a fully-featured OCaml backend for Lean or by using the Lean compiler's intermediate and later stages to compile code written in Rocq or other proof assistants.

Our contribution

We implemented a bridge from the Lean language to the $\lambda_{\square}$ language used in Rocq's verified extraction, adapting the erasure algorithm introduced by Letouzey (2004) to Lean expressions. The output of our implementation is suitable for compilation to executables using Rocq's existing verified extraction pipeline. In addition, we implemented some features handled specially by the Lean compiler, including fast computations on arbitrary-precision integers and multiple implementations of Lean's `Array` API, using external "axioms" written in OCaml and linked with the output of extraction.

We evaluated our work by measuring the execution times of a set of benchmark programs, based on the benchmarks introduced by Ullrich and de Moura (2021), compiled using the Lean compiler and Rocq's extraction pipeline applied to the output of our implementation of erasure. We performed measurements using several different configurations of the erasure and extraction pipeline in order to discern the effects of various configuration options and optimizations. Our results show that the executables produced by the extraction pipeline are on the order of 2.5 times slower, with substantial variation between the individual benchmarks. In particular, we show that the safe implementations of persistent arrays required when using OCaml's garbage-collected runtime incur a considerable slowdown compared to the Lean compiler's implementation, which uses reference counting to perform safe destructive updates.

Arguments supporting validity

We based our set of benchmark programs on the benchmarks used in the paper "Counting Immutable Beans" (Ullrich and de Moura 2021), which presents the reference counting optimizations performed by the Lean compiler and is the standard for evaluating the performance of Lean as a programming language. The adaptations necessary in order to avoid using Lean features not supported by our erasure step preserve the computation patterns of the benchmarks.

The main limitation of this work is the low number of benchmark programs and their relative simplicity; larger benchmark programs featuring substantial proof content and making use of idiomatic Lean programming styles would provide more realistic evaluations. Additionally, our implementation of erasure supports most of Lean's core language, enabling normal use of Lean's user-facing syntax in programs, but several features of the Lean language specially handled by the compiler are not supported, including I/O, asynchronous computation primitives and fixed-width integer types.

Summary and future work

We attribute the considerable margin by which the Lean compiler outperforms our extraction pipeline using OCaml to the combination of effective high-level optimization passes and dedicated reference counting optimizations at a lower level. The latter are not a common feature among functional programming languages, and hence using an existing compiler as a backend for Lean would not have been a better option during its development. The natural continuation of this work would be to develop a translation from the $\lambda_{\square}$ language towards an intermediate stage of the Lean compiler, which would allow it to be used to compile programs obtained by erasure from Rocq or other proof assistants.

Finally, the fact that Rocq's preexisting verified extraction pipeline provides the $\lambda_{\square}$ language as a clearly delimited, minimalist intermediate stage greatly simplified the process of adding a Lean frontend to the pipeline. If possible to do so without sacrificing performance, making Lean's compiler more modular would allow easier interfacing with alternative frontends or backends, enabling more diverse applications or the use of verified compilers to shrink the trusted computing base of verified Lean programs.

1. Introduction

The highest guarantee of software correctness is provided by formal methods for program verification, which verify mathematically that a program's behavior adheres to a formal specification. Verified software is free from bugs caused by incorrect implementation and hence, provided that the specification correctly describes the intended behavior, it is sure to never produce incorrect output or crash.

Proof assistant software such as Isabelle[1] (Nipkow et al. 2002), Lean (De Moura and Ullrich 2021), or Rocq[2] (The Rocq Development Team 2025), is one of many tools useful for formally verifying programs. Just like software verification enables high trust in programs, proof assistants provide formal guarantees about the correctness of mathematical proofs. Unlike informal, pen-and-paper mathematics, proofs formalized in a proof assistant are translated directly into the rules of the underlying logic and checked by a small trusted part of the proof assistant called the kernel. Proof assistants can be used to represent mathematical formalizations of the abstract syntax and semantics of programming languages, which allows properties of programs to be stated and verified. This is the approach used in the seL4 project (Klein et al. 2009), a general-purpose operating system microkernel whose C implementation is proven correct with respect to its abstract specification in the Isabelle proof assistant.

An advantage of using proof assistants over other formal methods for program verification is that their logical systems can express the entirety of mathematical reasoning: any mathematical property which is provable about a program can be expressed and proven within a proof assistant. This contrasts with methods based on abstract interpretation, where automated analysis is used to determine a conservative but not fully precise approximation of the program's behavior, or Satisfiability Modulo Theories (SMT) solvers, which are usually limited to properties expressible in first-order logic, without quantification over predicates or functions. The mathematical community has produced libraries of formalized proofs covering large parts of mathematics, including Isabelle's Archive of Formal Proofs (Eberl et al. 2004), Lean's Mathlib (The mathlib community 2020), and Rocq's Mathematical Components library (Mahboubi and Tassi 2022). Thus, advanced results which may be out of reach for automated tools can be used directly when proving statements about programs in a proof assistant.

The separation between the object-level programming language, in which the program to be verified is written, and the proof assistant's language, in which statements about the program are formulated and proven, can in fact be lifted. In proof assistants based on dependent type theory, including Agda[3] (Norell 2007), Lean and Rocq, both computational data, such as integers and lists, and logical data, such as statements and proofs, are represented as terms in a small functional language. This makes it possible to write and prove programs directly within the term language of the proof assistant, without an encoding layer between logic and programs. Moreover, proofs about a program's behavior can be partially or totally combined with its definition, using the type system to express hypotheses about well-formedness of data, information about unreachable cases or guarantees about the output. Writing programs in a high-level, purely functional language also simplifies reasoning about their behavior: the results of function calls depend only on the arguments passed, unlike in languages with mutability where function calls may depend on or modify shared state, and there is no need to reason about memory layout, ownership and aliasing, as is the case in languages such as C with pointers and manual memory management. Code written in type-theoretic proof assistants is also required to terminate on every input, unlike in other pure languages such as Haskell, which obviates the need to reason about nontermination. These considerations make reasoning about code written in the term language of proof assistants easier, sparing some of the complexity of tools, such as the Iris framework (Jung et al. 2018), which can express reasoning about mutation and exclusive ownership of various resources.

[1] Homepage: https://isabelle.in.tum.de/

[2] formerly named Coq

[3] Homepage: https://wiki.portal.chalmers.se/agda/pmwiki.php

However, implementing a type-theoretic proof assistant does not immediately yield a programming language usable for writing software. Proofs are verified as a part of type-checking, and so proving that a program written in a proof assistant is correct does not require the ability to actually execute any of its code! Proof assistants feature interpreters that can evaluate terms to normal forms, but these are not suitable for efficient execution of code, both because proof terms included in the code may require reduction steps and because of the run-time overhead of interpreters compared to compilation. To solve this problem, Paulin-Mohring (1989) and Letouzey (2004) introduced extraction as a method to obtain executable, machine-code programs from their implementation in a proof assistant. In this process, only computational content is preserved: the logical parts of the input program, which serve to prove correctness and do not infuence the program's computational behavior, may be safely discarded after the proof assistant has validated the input. The prime example of a large piece of verified software obtained via this method is the CompCert verified optimizing C compiler (Leroy 2009), which is written and proven correct entirely within the Rocq proof assistant, then extracted to OCaml for compilation.

The Lean and Rocq proof assistants have very similar term languages, but the mechanisms by which they both support transforming programs into executables differ significantly. In Rocq's extraction, programs are converted into a format understood by the OCaml compiler as early as possible, in order to rely on its optimization passes and garbage-collecting runtime. On the other hand, Lean's compiler performs several optimizing program transformations and handles low-level details such as reference counting operations itself, producing C code which is then compiled and linked with a dedicated runtime using a regular C compiler. In this internship, we compared these two approaches by implementing a bridge from Lean's term language into an early stage of Rocq's verified extraction pipeline, thereby enabling Lean programs to be compiled using the OCaml compiler as a backend. Comparison with the Lean compiler on a small set of benchmarks shows that binaries produced by extraction via OCaml are slower by a factor of about 2.5, with large variations between benchmark programs. We compare the design choices made in Rocq's extraction and Lean's compiler in light of these results, with the goal of informing future development of code extraction from proof assistants.

2. Writing and verifying code in type-theoretic proof assistants

We begin with a brief overview of the term language of proof assistants based on dependent type theory. For a more in-depth introduction to the features presented here, see Appendix A.

Dependent type theory, as used in Agda, Lean and Rocq, is essentially a strictly typed functional programming language similar to OCaml or Haskell. Functions are first-class objects and may be defined locally within expressions and passed to other functions. Aside from the function type `A -> B`, most common types are inductive types, whose values are generated by a finite set of constructors. Pattern-matching is used to destructure a value of an inductive type, retrieving the arguments passed to the constructor used to create it and branching evaluation depending on the constructor. Types themselves are also first-class values and may depend on other values: for example, the type of lists of length `n` depends on the number `n`. Functions whose return type depends on the argument passed have the dependent function type `(a: A) -> B(a)`, and pairs whose first element determines the type of the second have the dependent pair type `(a: A) × B(a)`.

Following the Curry-Howard correspondence between intuitionistic logic and type systems, the term language used in proof assistants can be used to encode logical reasoning. Under the propositions-as-types principle, a proposition is proven by exhibiting a term of the corresponding type. Complex propositions can be built using type formers: for example, implication is represented by function types, and universal and existential quantification are expressed using dependent function resp. pair types.

The features of the term language used to represent computations and logic are shared, which enables it to be used to write programs which combine computational code together with logical parts expressing

invariants, well-formedness of data, or adherence to a specification. As an example, the following toy program, written using Lean syntax, bundles together the computation of a^k, where a and k are natural numbers and a is assumed to be nonzero, with the proof that the result is also nonzero:

```
def pow (a: Nat) (ha: a > 0) (k: Nat): { n : Nat // n > 0 } :=
  match k with
  | .zero => ⟨1, Nat.one_pos⟩
  | .succ k' =>
    let ⟨(m: Nat), (hm: m > 0)⟩ := pow a ha k';
    let n: Nat := a * m;
    let hn: n > 0 := Nat.mul_pos a m ha hm;
    ⟨n, hn⟩
```

In addition to the two computationally relevant arguments `a` and `k`, the function `pow` also explicitly takes the hypothesis `ha`, which is evidence of the proposition stating that `a` is nonzero, as an argument. Its return value has the subset type `{ n : Nat // n > 0 }` and consists of a dependent pair whose first element `n` holds the result of the computation and whose second element is evidence that `n` is nonzero. Thus, the requirements on the input as well as the guarantees provided on the output of `pow` are included directly in its type signature, and the fact that the type checker accepts the implementation guarantees that it conforms to the provided specification. Within the body of `pow`, we use pattern matching on the argument `k` to distinguish between the two possible cases for a natural number: either `k` is zero, in which case a^k is 1, or `k` is the successor of some `k'`, in which case a^k is equal to $a \times a^{k'}$. The value of $a^{k'}$, together with evidence `hm` proving that it is nonzero, is obtained by a recursive call to the function `pow`. The proof by induction that the result is positive is combined with the recursive definition of exponentiation, using the two lemmas `Nat.one_pos: 1 > 0` and `Nat.mul_pos: (a: Nat) -> (b: Nat) -> (a > 0) -> (b > 0) -> (a*b > 0)` from Lean's standard library in order to conclude in both the base case and induction step.

As the example of `pow` shows, the term language of theorem provers is suitable both for writing programs and for expressing reasoning about their behavior, with a type system ensuring the validity of proofs. We now turn to the way Lean and Rocq programs are converted into executable code.

3. Obtaining executable code from Rocq and Lean

3.1. Common parts

The surface language presented to users of proof assistants is much richer than the frugal fragment demonstrated above, with common conveniences including

- An extensible notation system which introduces new ways to write and print expressions
- Marking certain function arguments as implicit arguments
- Matching on multiple discriminants with flexible patterns
- Typeclasses, which provide support for ad hoc polymorphism
- A tactic language, which allows proof terms to be constructed interactively
- Top-level commands, which may have a wide range of effects, such as printing the definition of a term, configuring the way the rest of the file will be processed, or running metaprograms.

After the syntax of the surface language is parsed, it is processed in a step called elaboration, during which tactics and commands are run and implicit information is inferred, and converted into a list of declarations. Declarations include inductive type declarations, which introduce a new inductive type former and its constructors, constant definitions, which associate an expression in the term language with a name which may later be used to refer to it, and axioms, which introduce a name with a certain type ex nihilo. Declarations are sent in order to the proof assistant's kernel for type-checking, after which they are added to the global environment, a set of valid declarations which can be used in

subsequent declarations. Keeping the proof assistant's kernel separate from the elaborator allows users to interact with a rich and convenient layer, while type checking is only implemented for a simple term language covering only the core constructs of the type theory used, thus reducing the surface area for soundness bugs which would lead to the system accepting an incorrect proof.

In dependent type systems, type checking may require partial evaluation of expressions, as the well-typedness of dependently typed programs can depend on the result of computations. However, the interpreters used for this purpose are impractical for evaluating large programs, both because of interpretation overhead and because subterms containing purely logical content may also require evaluation.[4] Therefore, after validation by the type checker, declarations representing actual computations which should be executed must undergo additional processing in order to yield an executable program. Since the type system forbids any computational value from depending on the value of a proof, all logical content can be removed from a program and replaced with uninformative dummy values, which is known as proof erasure. The resulting terms are then converted into natively executable binaries. While the preceding steps are mostly analogous in all type-theoretic proof assistants, from this point on, Lean and Rocq adopt two conceptually distinct approaches, compilation and extraction.

In extraction, the goal is for the steps implemented as part of the proof assistant to do as little work as possible, transforming the term language into source code for a standard high-level functional programming language, such as Haskell or OCaml. Functional constructs in the term language, such as local function definitions or partial application of curried functions, can be translated directly by using the corresponding syntax in the target language. Aside from considerations related to differences between the proof assistant's and the target language's type systems, which are resolved by inserting casts, extraction essentially amounts to pretty-printing the term language with the appropriate syntax. The rest of the path down to executable binaries is handled by the existing compiler for the target language. By contrast, compilation refers to translation into a low-level language, such as assembly or C, in which high-level functional constructs are not available. Instead of relying on the target language's compiler to perform sensible optimizations, as is the case in extraction, a proof assistant performing compilation applies various programs transformations, aiming to improve runtime performance, during the intermediate stages used to lower terms into the target language.

Several considerations influence the choice between compilation and extraction. Opting for extraction greatly reduces the amount of code which must be written and maintained by the developers of the proof assistant, since the implementation of a runtime system and the fine-tuning of optimization passes are left to the development team of a high-level programming language's compiler. Moreover, using extraction can also increase trust in the system, since the smaller codebase is easier to review or even formally verify than an entire compiler included within a proof assistant. Extraction and compilation both require trust in the compiler of the target language, which may introduce miscompilations (Yang et al. 2011) despite regular testing; verified compilers such as CompCert (Leroy 2009) or CakeML (Kumar et al. 2014) provide a formal guarantee of correctness. Despite the cost in development effort and increased trusted computing base size, implementing a bespoke compiler for a proof assistant can provide performance benefits, as the optimization passes can be designed around specific idioms present in the term language such as typeclasses and monadic programming, whereas off-the-shelf compilers may have more difficulty efficiently compiling these features in extracted code. Finally, the foreign function interface available in the target language or supported by the compiler are revelant to interoperability with other languages, which is commonly used to write and verify the core of an application within a proof assistant and implement a user interface in another programming language.

We will now present Lean's compiler and Rocq's verified extraction pipeline in more detail.

[4] The need to evaluate propositions depends on the specifics of a proof assistant's underlying type theory. In Lean, all propositions are proof-irrelevant; Rocq's universe `SProp` is proof-irrelevant, but the more commonly used `Prop` is not.

3.2. The Lean 4 compiler

Lean 4 (De Moura and Ullrich 2021) is the latest iteration of the Lean proof assistant, introducing a compiler to C which enables Lean to be used as a general-purpose programming language. Invoking Lean's compiler and compiling and linking the produced C code with Lean's language runtime (using either GCC or Clang, depending on the platform) is usually performed using Lean's build system, Lake, which provides a simple workflow for users. Another feature simplifying Lean's use for programming is its support for I/O operations, including reading standard input and output and manipulating files, which are usually unavailable in the pure term languages of proof assistants. Lean integrates these side-effects in the style of Haskell, using an IO monad (Wadler 1997) to manipulate the description of effectful actions within pure code. As a testament to its usability for writing real programs, large parts of the Lean 4 system are in fact bootstrapped and implemented in Lean itself, including the entire parser and elaborator, and, as of the newly-released version 4.22, the entire compiler as well.[5]

Because it will be the starting point for our bridge between Lean's compilation pipeline and Rocq's extraction, let us now take a closer look at Lean's term language as it is output after elaboration.

```
inductive Expr where
  | lit (l : Literal)                                    | bvar (deBruijnIndex : Nat)
  | sort (u : Level)                                     | fvar (fvarId : FVarId)
  | const (declName : Name)                              | app (fn : Expr) (arg : Expr)
  | proj (typeName : Name) (idx : Nat) (struct : Expr)
  | lam (binderName : Name) (binderType : Expr) (body : Expr)
  | forallE (binderName : Name) (binderType : Expr) (body : Expr)
  | letE (declName : Name) (type : Expr) (value : Expr) (body : Expr)
```

Listing 1: Simplified definition of the type `Expr` in Lean, representing expressions in the term language. Some constructors and arguments have been omitted for brevity.

The abstract syntax of Lean's term language is represented by an inductive type, `Expr`, shown in Listing 1. The simplest leaves of the expression tree are the constructors `.lit` representing literals of type `Nat` or `String`, `.sort` which represents the special types `Prop` and `Type`, and `.const`, which is used to refer to a value added to the global environment by a preceding declaration using its fully qualified name. Binary application of a term to another is represented by `.app`, and the constructor `.proj` represents access to a field of a structure – a nonrecursive inductive type with a single constructor – by its numerical index. The constructors `.lam`, `.forallE` and `letE` represent the different ways to bind local variables: lambda-abstraction, as in the term `fun x: X => _`, dependent function types, as in the term `(x: X) -> _`, and let-bindings, as in `let x: X := _; _`. The variables bound by these constructs are represented by the constructors `.bvar` and `.fvar`, following the locally nameless approach (McBride and McKinna 2004): variables which are bound within a term are represented using de Bruijn indices, and free variables are represented by a unique string identifier. Terms involving bindings are constructed and destructured using an API which automatically handles the transformation between bound and free variables, which avoids the hassle of handling de Bruijn indices manually and issues due to variable name shadowing while ensuring that no free variables are present in closed terms.

`Expr` does not explicitly represent constructors, pattern matching nor recursion, but rather encodes them using the constructs presented above. Constructors are added to the global environment after Lean's kernel processes the declaration of the inductive type they belong to, and referred to by name using the constructor `.const` just like any top-level definition. Complex pattern matching, for which user-facing syntax significantly more expressive than what was shown in Section 2 and Appendix A is available, is converted by the elaborator into calls to auxiliary matcher functions, which take the discriminees to match on and functions describing the computation to run in each arm of

[5] Lean's kernel, CLI interface, interpreter and runtime are still written in C++.

the `match` expression as arguments. The auxiliary matcher functions themselves are defined using `casesOn`, nonrecursive dependently typed eliminators which encode reasoning by cases on single values of inductive types. For example, the eliminator for the type `Nat`, named `Nat.casesOn`, has type `(P: Nat -> Type) -> (P 0) -> ((m: Nat) -> P (m+1)) -> n -> P n`, expressing the fact that a term of type `P n` can be constructed for any `n`, given a term of type `P 0` and a term of type `P (m+1)` for any `m`. Definitions for the `casesOn` functions are produced by the elaborator automatically when it processes an inductive type declaration, and are referred to using `.const` like constructors.

Recursion is a point of divergence in the Lean pipeline and is handled differently for compilation and type-checking. First, during regular elaboration, term-level local recursion using the `let rec` syntax is lifted to recursion between top-level functions. Mutually dependent top-level definitions are grouped in mutual blocks, which are processed together and allow recursion. This format is adequate for code generation, since C supports mutual recursion using forward declarations, and so definitions are sent to the compiler in this form after type-checking without further processing. The terms sent to the kernel undergo an additional elaboration step in which recursion between top-level functions is eliminated and replaced with recursors, primitive features of Lean's type theory which encode structural recursion, generalizing the way `casesOn` functions express case analysis. This obviates the need for termination checking inside the kernel, thereby increasing Lean's robustness as a proof assistant.

Having obtained an overview of `Expr`, we now turn to Lean's compiler, which converts definitions from `Expr` into C. The first intermediate format used by the compiler is LCNF[6], based on A-normal form (Flanagan et al. 1993), in which the expression tree is replaced by an ordered sequence of basic operations. Several standard optimization passes are performed on LCNF, including join point insertion (Maurer et al. 2017), lambda lifting, removal of unused function arguments, function specialization and expression simplifications. Next, LCNF is lowered into a format named IR[7], in which reference counting operations are explicitly represented. A number of optimizations regarding reference counting are applied to IR, including marking certain function parameters as borrowed and detecting opportunities for the generated code to perform destructive updates, instead of requiring new allocations, by dynamically detecting situations where the last reference to an object is dropped in close proximity to the creation of a new object of the same kind (Ullrich and de Moura 2021). Finally, IR is converted into C code, essentially by printing. Proofs of propositions are removed during the conversion of `Expr` into LCNF, but information about the types of terms is retained in order to support the special handling of certain types by the compiler and runtime. These include fixed-width integer types such as `UInt32`, the types `Nat` and `Int` which are logically represented using a unary encoding but compiled to use the GMP arbitrary-precision arithmetic library (Granlund and the GMP development team 1991) for efficiency, and the type `Array α`, which is logically represented as a linked list but compiled as a dynamic array.

3.3. Rocq's verified extraction

The Rocq Prover's extraction mechanism, introduced by Paulin-Mohring (1989), supports extraction from its term language Gallina to OCaml, Haskell and Scheme, with the modern implementation based on Letouzey's PhD thesis (2004). Recently, Forster, Sozeau, and Tabareau (2024) have built upon this work to provide a formally verified extraction pipeline, which will be the one presented in detail here. The verified pipeline begins with type and proof erasure, which removes types and logical content by translating Gallina into a minimal untyped λ-calculus named $\lambda_{\square}$ (read "lambda-box"). This step, described in Letouzey's thesis, was implemented and formally proven in Rocq as part of the MetaRocq project (Sozeau et al. 2025). Erasure is followed by conversion of $\lambda_{\square}$ into the Malfunction language, also formally verified in Rocq, and so Malfunction's compiler is the only step in the pipeline requiring trust. We now describe the syntax of $\lambda_{\square}$, the target for our bridge into Rocq's verified extraction pipeline.

[6] Lean Compiler Normal Form, referred to as λ_{pure} by Ullrich and de Moura (2021).

[7] Intermediate Representation, referred to as λ_{RC} by Ullrich and de Moura.

```
inductive LBTerm where
| const: Kername -> LBTerm                                  | box: LBTerm
| app: LBTerm -> LBTerm -> LBTerm                           | prim: PrimVal -> LBTerm
| proj: ProjectionInfo -> LBTerm -> LBTerm                  | bvar: Nat -> LBTerm
| lambda: BinderName -> LBTerm -> LBTerm                    | fvar: FVarId -> LBTerm
| letIn: BinderName -> LBTerm -> LBTerm -> LBTerm
| construct: InductiveId -> Nat -> List LBTerm -> LBTerm
| case: (InductiveId × Nat) -> LBTerm -> List (List BinderName × LBTerm) -> LBTerm
| fix: List (@FixDef LBTerm) -> Nat -> LBTerm

inductive GlobalDecl where
| inductiveDecl (body: MutualInductiveBody)       | constantDecl (body: ConstantBody)

abbrev Program: Type := (List (Kername × GlobalDecl)) × LBTerm
```

Listing 2: Abstract syntax for $\lambda_\square$ expressions and programs, as ported to Lean for this project. The constructor `.fvar` is an implementation detail not included in the original definition in MetaRocq.

Listing 2 shows the abstract syntax of $\lambda_\square$. A program is a pair containing a global environment, represented as a list of declarations which are either definitions or inductive type declarations, and a term. Although $\lambda_\square$ is untyped, information regarding how many constructors an inductive type has and how many arguments they each take must be kept for the translation to Malfunction. The syntax of $\lambda_\square$ terms consists mostly of standard features of lambda-calculus, including variables represented by de Bruijn indices, λ-abstraction, let-binding, and binary application. Just like Lean's `Expr`, $\lambda_\square$ includes projections to access structure fields, constants to access names in the global environment, and primitive values. Constructors and case analysis are represented explicitly in $\lambda_\square$ using `.construct` and `.case`. Since $\lambda_\square$ represents terms after erasure, there are no constructors representing the sorts `Prop` and `Type` or the dependent function type. Instead, all erased information is replaced by a special term □, represented by the constructor `.box`.[8] Mutually recursive function definitions are primitive in $\lambda_\square$ and explicitly represented at the term level, using the constructor `.fix` to introduce multiple local function definitions at once, each of them available within the bodies of the others, then select one.

Rocq's verified extraction converts $\lambda_\square$ programs to Malfunction (Dolan 2016), a minimal programming language very close to the OCaml compiler's intermediate representation and designed as an easy target for compilation from high-level functional languages. Malfunction is an untyped language, and so Rocq's verified extraction, unlike the regular extraction to the OCaml surface language, does not use the unsafe casting function `Obj.magic`, which can inhibit optimizations (Pottier 2001); program safety, including absence of segfaults, is left entirely within the responsibility of Rocq's type system.[9] Malfunction is compiled using the OCaml compiler backend, including the Flambda optimizer, assembly code generation, and fast and precise garbage collector, and therefore code extracted from Rocq using the verified pipeline can be linked with other programs written in OCaml or use a C FFI.

The verified extraction pipeline is distributed as a Rocq plugin which outputs Malfunction source code directly after being invoked inside a Rocq source file. The $\lambda_\square$ language is only a formal intermediate step, and not exposed as an output or input format. However, because $\lambda_\square$ corresponds closely to the term language of any dependently typed proof assistant with proofs and types removed, it has the potential to become a common middle point for modular extraction pipelines based on frontends from various proof assistants to $\lambda_\square$ and backends from $\lambda_\square$ to several target languages. The `lbox` tool (Nielsen and Spitters 2025) implements the back half of this factorization as a self-contained executable which parses a simple S-expression representation of $\lambda_\square$ and outputs code in a variety of programming

[8] Propositional content cannot be removed entirely by erasure when targeting a strict language such as Malfunction.

[9] Details related to Rocq's dependent type system and OCaml's support for mutability limit the correctness theorem for verified extraction to first-order interfaces, through which extracted code can safely interact with arbitrary OCaml code.

languages including Wasm, Rust, C, Elm, and Malfunction.[10] The Malfunction backend for `lbox` is based on Rocq's extraction pipeline and constitutes the entry point we need: we will serialize $\lambda_{\square}$ programs obtained from Lean `Exprs` to strings and convert them to Malfunction using the `lbox` tool.

4. Implementing erasure from Lean to $\lambda_{\square}$

4.1. Basics

We extract expressions from Lean as fully elaborated `Exprs`, just before they would be sent to the compiler, and convert them to $\lambda_{\square}$ before entering Rocq's verified extraction pipeline. The similarity of `Expr` and $\lambda_{\square}$ makes them a convenient pair of source and target language: differences in the structure of expressions notwithstanding, the transformation from `Expr` into $\lambda_{\square}$ is conceptually quite simple and corresponds to the type and proof erasure step in Rocq's verified extraction pipeline.

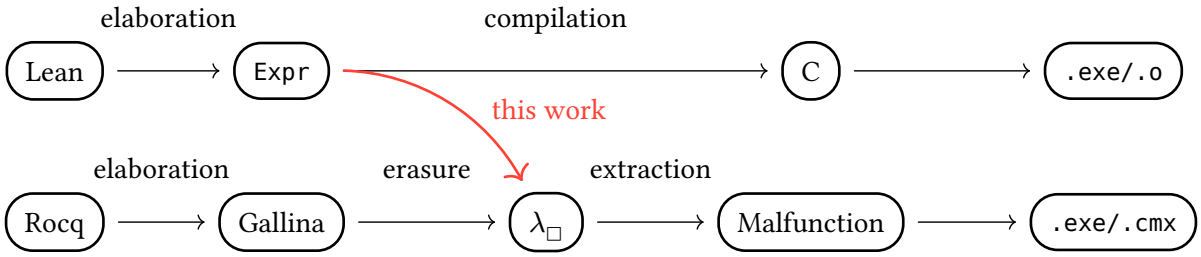

Figure 1: Position of this work as a bridge between the Lean and Rocq pipelines.

Our erasure from `Expr` to $\lambda_{\square}$ is implemented in Lean as a function taking an input of the type `Expr` shown in Listing 1 and producing a term of the type `LBTerm` shown in Listing 2. In order to handle references to global constants and inductive types, the erasure procedure must be able to read from Lean's global environment and produce a $\lambda_{\square}$ global environment, i.e. a list of $\lambda_{\square}$ declarations, alongside the erased term. These effects are supported using a dedicated monad, `EraseM`, as the context for functions related to erasure. Additionally, `EraseM` manages a local context, mapping the identifiers of variables to their type and optional value (for let-bound variables). This allows the type of subterms featuring free variables to be determined, which is used to determine whether they can be erased. A small collection of helper functions working in `EraseM` are used to automatically manage free and bound variables following the locally nameless approach while destructuring `Expr` terms and constructing $\lambda_{\square}$ terms. Using these helper functions, the general process of converting an `Expr` to a $\lambda_{\square}$ term is straightforward: terms whose type is either of the form `_ -> ... -> Type` (indicating that the term is a type former) or whose type is a proposition are directly converted to `.box`, and non-erasable terms are destructured into component `Exprs` using the appropriate helper function. These are then recursively erased into `LBTerms` and reassembled using the matching construction function. However, the differences between `Expr` and `LBTerm` add complications to this basic scheme.

For instance, recall that in `Expr`, constructors are top-level constants and case analysis is represented using special constants named `casesOn`, whereas $\lambda_{\square}$ has dedicated syntax for these features. In order to handle partial applications of constructors or `casesOn`, which are allowed in Lean but not in $\lambda_{\square}$, it is necessary to know how many arguments a constant is applied to when processing it. This requires special traversal of binary function applications in the expression tree. For example, the expression `a b c`, represented as `.app (.app a b) c`, will be considered as `a` applied to the two arguments `b` and `c`, instead of recursing immediately to process `.app a b` and `c` separately. If `a` is a constructor taking three arguments, then it is underapplied in this example, which is resolved by adding additional arguments via η-expansion, e.g. handling the expression `a b c` as `fun d => a b c d`. Once all arguments are present, constructors and `casesOn` functions are translated using the `.construct` resp.

[10] For Rust and Elm, which are typed languages, a variant of $\lambda_{\square}$ named $\lambda_{\square}^{T}$, which keeps more type information, is used.

`.cases` constructors of `LBTerm`, with some additional care required for match arms, which are passed as functions in `Expr` and as open terms whose free variables are bound by the `.cases` syntax in $\lambda_{\square}$.

Whenever the erasure procedure encounters a term constructing or deconstructing a value of an inductive type for the first time, it adds the corresponding inductive type declaration to the $\lambda_{\square}$ global environment managed by `EraseM`. Similarly, instances of `Expr`'s `.const` constructor which do not require any special treatment are normal references to top-level constant declarations. Upon encountering a constant name for the first time, the erasure procedure looks up its declaration in Lean's global environment, performs erasure on the definition body, then adds a declaration with the name and erased definition to the $\lambda_{\square}$ global environment. Mutually recursive definitions present in `Expr` are handled by processing all declarations in a mutual block together, using `EraseM` to remember the names of the functions being defined, and creating independent $\lambda_{\square}$ declarations, each representing the auxiliary definitions at the term level using the `.fix` constructor of `LBTerm`. Once a constant has been added to the $\lambda_{\square}$ global environment, it is referred to using the `LBTerm` constructor `.const`.

Another aspect complicating semantically-correct erasure from Lean to $\lambda_{\square}$ is control flow. While `LBTerm`'s `.cases` behaves like Lean's `casesOn` and only evaluates only the chosen branch, functions used to choose between alternatives will evaluate all of their arguments strictly. For example, the following function is used in Lean as the implementation of if-then-else syntax for boolean conditions:

```
def cond (α : Type) (c : Bool) (x y : α) : α :=
  match c with
  | .true  => x
  | .false => y
```

Using the `cond` function naïvely in programs can introduce nontermination because Lean is a strict language: evaluating the function call `cond .true a b` will always fully evaluate `a` and `b`. Lean resolves this by marking `cond` and similar functions with the `@[macro_inline]` attribute, which indicates that their definition should be unfolded at every call site before compilation. We handle this as part of a preprocessing pass run on expressions before erasure. Besides `@[macro_inline]` functions, this pass also inlines auxiliary matcher functions produced by Lean's pattern matching elaboration and appends the suffix `_unsafe_rec` to the names of recursive functions in order to access their original definitions, which use toplevel recursion, instead of the result of Lean's recursor elaboration.[11] Finally, the preprocessing step applies Lean lemmas tagged `@[csimp]`, which direct Lean's compiler to replace the logical definition of a function with an equivalent implementation. This is crucial for performance, as many common functions have tail-recursive implementations provided by `csimp` lemmas.

The steps described above, including preprocessing of `Expr` terms, type and proof erasure of a term and recursive erasure of all required declarations, suffice to transform an `Expr` within its Lean global environment into a $\lambda_{\square}$ program. The erasure is made available via a Lean module named `Erasure`, which defines a new toplevel command `#erase <term> to <path>` that fully elaborates `<term>` into an `Expr` which is erased to `LBTerm`; the resulting program is then serialized to a string using the S-expression syntax expected by the `lbox` tool and written to the filesystem:

```
import Erasure

def val_at_false (f: Bool -> Nat): Nat := f .false

#erase val_at_false to "out.ast"
```

Elaborating this Lean code produces a file `out.ast` containing the $\lambda_{\square}$ code for the program `val_at_false`, which can then be converted to Malfunction code using the `lbox` tool, compiled to

[11] With one exception discussed below, we assume that recursors do not appear in terms prior to recursion elaboration.

a `.cmx` object file with the Malfunction compiler, and linked with other OCaml object files using `ocamlopt`, following the same steps as Rocq's verified extraction pipeline. While the erasure procedure presented is sufficient to compile simple programs written in Lean, we implemented several additional features supported by Lean's compiler in order for the benchmarks to be a more fair comparison.

4.2. Axioms and arithmetic

The Lean language provides an `@[extern]` attribute which directs the compiler to use a C definition to implement a given function.[12] Function declarations tagged `@[extern]` may optionally have a definition written in Lean, in which case the Lean definition is used by the type checker and the external C definition is used by the compiler.[13] Lean uses this mechanism extensively to override simple logical definitions with efficient code written in C. Most centrally, the type `Nat` of natural numbers is defined in Lean as an inductive type following Peano arithmetic (see Appendix A). If `Nat` were to be compiled normally, representing the number n would require $\Theta(n)$ space on the heap, and addition, multiplication and exponentiation of natural numbers would have time complexities linear in the size of the result, which is unsuitable for programs manipulating numbers of any significant size. The Rocq prover, which uses an equivalent encoding for its type `nat`, works around this issue by including types `N` and `Z`, which provide a binary representation of natural numbers resp. integers, in its standard library.[14] Operations on `N` and `Z` run in time proportional to the size of a number in bits, which provides much better performance than `nat` in code destined for extraction, but conversions between the isomorphic `N` and `nat` are a frequent inconvenience. In Lean instead uses a single type `Nat` (resp. `Int` for integers) whose logical representation is overridden by the compiler, which represents small numbers directly as machine integers and larger numbers using an arbitrary-precision arithmetic library. The basic arithmetic functions are tagged `@[extern]`, replacing their recursive logical definitions with C implementations included in the Lean runtime, and the C definitions are marked for inlining, in order to remove function call overhead for common operations such as addition.

We added support for efficient arithmetic operations to our erasure to $\lambda_{\square}$ by using OCaml's `Zarith` library (Miné, Leroy, and Cuoq 2011), a GMP wrapper very similar to Lean's runtime representation of integers, to represent values of type `Nat` and `Int`. Since using `Zarith` alters the memory representation of values of type `Nat`, every point in Lean code where natural numbers are created or destructured must be rewritten. Since `Zarith` represents small integers identically to OCaml's native 63-bit signed integers, which also underly the primitive integers available in $\lambda_{\square}$, literals of type `Nat` found in an `Expr` can be transformed directly into a $\lambda_{\square}$ primitive integer provided they do not overflow this size bound. The constructors `.zero` and `.succ` are replaced by the literal `0` and the function `fun n => n + 1`, and the case analysis performed by `Nat.casesOn` is replaced by an expression of the form `if n == 0 then … else …(n-1)`, explicitly testing whether a number is zero and subtracting one to obtain the predecessor if not. In order to provide efficient `Zarith` implementations for the arithmetic functions marked `@[extern]` in Lean, the erasure procedure adds a declaration without a body, called an axiom, to the $\lambda_{\square}$ environment upon encountering a constant declaration which either has no associated value or is marked with the `@[extern]` attribute.

Axioms present in a $\lambda_{\square}$ program processed by the `lbox` tool will become references to an OCaml module named `Axioms` during conversion to Malfunction. This module can be provided by writing an `axioms.ml` file in OCaml and linking it with the `.cmx` file produced by the Malfunction compiler in order to produce the final executable. In addition to efficient implementations of arithmetic functions on `Nat` and `Int`, which are implemented using `Zarith` and marked for inlining by the OCaml compiler,

[12] The attribute `@[export]` allows the reverse, exposing a function defined in Lean to C code.

[13] This is a source of unsoundness if the external implementation of the function does not match the logical definition.

[14] https://rocq-prover.org/doc/V9.0.0/corelib/Corelib.Numbers.BinNums.html

we use axioms to provide definitions for the equality type's recursor `Eq.rec`[15] and operations on Lean's `Array` type, whose implementation we discuss in the next section.

4.3. Efficient functional arrays

Lean's `Array α` type is a data structure containing an ordered sequence of elements of the same type `α`. The operations on `Array` are similar to dynamic arrays in programming languages featuring mutability, such as Python's `list` and OCaml's `Dynarray`: elements of an array can be read and set at an arbitrary index, and the array's size may be changed by adding or removing elements at one end. However, the Lean programming language does not feature mutability, and hence all operations that would modify an array in-place in a conventional programming language must instead return a new array in Lean:

```
def a: Array Nat := #[0, 1, 2]
def b: Array Nat := a.set 0 42
#check (rfl: a[0] = 0)
#check (rfl: b[0] = 42)
```

Setting the entry at index 0 in the array `a` to 42 does not modify `a` but rather creates an entirely new array `b` holding the result; after the `set` operation is performed, the value of `a` at index 0 is still 0.

Mutable dynamic arrays, laid out contiguously in memory, have excellent performance: random-access reads and writes are constant-time, consisting essentially in a pointer access with the desired offset, and the reallocations necessary to grow and shrink the array take amortized $O(1)$ time over a large number of operations. However, the `set` operation on such an array is inherently destructive, and so it cannot match the semantics expected of arrays in purely functional languages, which must be persistent, unless a full copy of the array is performed before any modification, which incurs $O(n)$ runtime cost.

The dilemma between persistence and good performance has spawned a number of solutions to the problem of arrays in functional programming languages. Tree-like data structures, such as Clojure's `PersistentVector`, can provide $O(\log_b(n))$ access and updates, which is not meaningfully distinguishable from constant-time updates with the typical value $b = 32$. Alternatively, in what is known as Baker's trick, mutable references may be used to store the chains of modifications applied to an array, providing an interface through which the data may be accessed by multiple users in a fully persistent way while the underlying mutation allows successive updates by a single user to be performed in constant time (Aasa, Holmström, and Nilsson 1988). Finally, it is sometimes possible to skimp on persistency, such as in cases where only a single reference to the latest version of an array is kept and the original version is never reused after an update. The PVS verification system performs a conservative static analysis on code featuring arrays in order to detect such situations at compile-time and produce code performing destructive array updates (Shankar 1999). Ensuring that a given program uses arrays without ever reusing an array which has been updated can also be done at the type level, using linear or ownership-tracking type systems to have the `set` function consume or invalidate the old reference to an array.

The Lean compiler leverages its reference-counting runtime system to provide an array datatype which has excellent performance for typical use-cases and yet behaves entirely persistently. Although Lean's `Array α` is defined as a trivial wrapper around `List α` for logical purposes, the compiler overrides the associated constructors and API functions with C implementations included in the Lean runtime, which represent arrays as reference-counted dynamic arrays. The C code for update operations such as `Array.set` inspects the reference count of the passed array at runtime. If the reference passed to the `Array.set` function is the only existing reference to the array, then the buffer may be modified in-place, providing $O(1)$ updates of unshared arrays; if the array is pointed to by additional references,

[15] `Eq.rec` is used by Lean's `match` elaborator for type casts within match arms. and becomes the identity at runtime

the existing buffer may not be mutated, and hence it is copied, which results in a new, uniquely owned array. This behavior is entirely invisible to user code.

The $\lambda_\square$ code produced by erasure is compiled via Malfunction and uses the OCaml runtime, which relies on a garbage collector for memory management, unlike Lean's reference-counting runtime. This precludes imitating the Lean compiler's approach of copying arrays only when this might be observable and destructively updating otherwise. Instead, we provide several implementations of Lean's `Array` API, to be included in `axioms.ml`. The first implementation uses the OCaml library `Sek` (Pottier and Charguéraud 2020), which uses a tree-like data structure, and the second is based on an OCaml implementation of persistent arrays using Baker's trick (Conchon and Filliâtre 2007), extended to allow for push and pop operations. In order to estimate the performance overhead of using persistent arrays, we also ran benchmarks with an unsafe implementation of `Array` based on OCaml's mutable `Dynarray`; executables produced in this way will silently produce incorrect results or even crash if the persistence of arrays is relied upon. Fortunately, idiomatic Lean tends to eschew shared references to arrays in order to avoid the $O(n)$ cost of copies, and the benchmarks used to compare the performance of array implementations run correctly.

4.4. Constructor pruning and unboxing

The Lean compiler guarantees that trivial wrappers around a type are represented in memory identically to the type itself, without indirection. More precisely, trivial wrappers are inductive types with exactly one constructor, which itself has only one computationally relevant argument and any number of irrelevant arguments. For example, the type `Fin n` represents natural numbers strictly smaller than `n`; its constructor, `Fin.mk`, takes one argument `val` of type `Nat` and one argument `isLT` which is a proof of the proposition `val < n`. Lean's compiler represents values of `Fin n` identically to their field `val`, so creating and destructuring `Fin n` values are no-ops and do not require allocation or following pointers, which makes bundling proofs and data together free in terms of runtime performance.

In order to perform this optimization using the Lean to $\lambda_\square$ to Malfunction pipeline, it is split into two steps, constructor pruning and unboxing. In constructor pruning, which is done as part of the erasure from `Expr` to $\lambda_\square$, all computationally irrelevant arguments of constructors are removed. This affects the inductive type declarations added to the $\lambda_\square$ global environment, every application of a constructor, and every match arm. Pruning must be done as part of erasure and cannot be performed later in the pipeline, as a $\lambda_\square$ program no longer retains information about the types of constructor arguments, which is necessary in order to remove irrelevant arguments. Next, unboxing consists in recognizing inductive types with a single one-argument constructor and removing occurrences of the constructor and case analysis. This is done in $\lambda_\square$, which is the latest possible stage for this step, as programs contain no inductive type declarations in Malfunction, which makes it impossible to tell the difference between an inductive type with a single constructor and a type with multiple constructors of which only one is used. Performing unboxing as a part of erasure would also be a possibility, but we chose to simply patch the `lbox` tool in order to set a flag enabling unboxing in MetaRocq's translation of $\lambda_\square$ to Malfunction.

When constructor pruning and unboxing are enabled, some parts of the `axioms.ml` file must be adapted in order to match the different memory representation. For example, the Lean type `Decidable P`, which is equivalent to a boolean carrying either evidence that the proposition `P` is holds or evidence that the negation of `P` holds, has two constructors `.isTrue` and `.isFalse` with a propositional argument which is removed by pruning. Since `Decidable P` is used in several `@[extern]` functions such as `Nat.decEq` which are implemented in `axioms.ml`, the OCaml type used as the return type for the implementations of these axioms must be adjusted to include a dummy argument for the logical evidence when pruning is disabled and remove the dummy argument when pruning is enabled.

5. Benchmarks and results

Having obtained a way to turn Lean code into executables by passing the output of erasure to the `lbox` tool and the Malfunction compiler, we now turn to the question of measuring the performance of executables thus produced and comparing it to executables produced using the regular Lean compiler. The source code for our implementation of erasure and benchmarks is available on GitHub.[16]

5.1. Test methodology

The source code for all benchmarks is written in Lean. Because the erasure pipeline does not support Lean's `IO` actions, the benchmark functions themselves, which must be compilable both using the Lean compiler and using extraction to $\lambda_{\square}$, cannot perform any side effects, including input or output. The fact that the benchmarks are pure functions makes naïve measurements of their execution time less meaningful, as a sufficiently advanced but correct compiler may choose to precompute the result of pure computations with a known input, or conversely, eliminate pure computations whose result is unused as dead code. Therefore, while the logic of each benchmark is shared between the Lean and Malfunction backends and kept as a pure function of type `Nat -> Nat`, the benchmarks are turned into executables by including each of them inside a minimal wrapper, written in Lean resp. OCaml, which reads an input number from the command-line arguments passed to the executable, evaluates the benchmark function on that number, and prints the result to standard output. This prevents the Lean and OCaml compilers from optimizing away the very computations we seek to measure.

Another consideration when reporting a speedup factor between compilation via the Lean compiler and extraction via Malfunction is that the executables generated by the Lean and OCaml compilers have nonzero startup times: even an empty program takes some amount of time to run, typically a few milliseconds, due to overhead generated by dynamic linking and setup required by the language runtime. We adjust for the constant startup penalty by running the benchmarks on large values of the argument n, in order to minimize the relative impact of startup noise, and we subtract the execution time of a baseline run with $n = 0$ before computing ratios in order to compare the execution time of only the phases in which the actual benchmark code is run.

Execution time of the test binaries produced was measured using the `hyperfine` tool (Peter 2024). All measurements were performed on a machine with an Intel i5-7200U CPU and 16GB of RAM. At the problem sizes chosen no swapping of memory pages to disk occurred. The `hyperfine` process and benchmarked program itself were run on a CPU core isolated using the `isolcpus` Linux kernel parameter. Hyperthreading, Intel Turbo Boost and CPU idle states were left enabled, as preliminary testing indicated these options did not significantly affect variance in benchmark execution times. The CPU frequency remained stable at 3100MHz without thermal throttling during all tests.

5.2. Backend configurations tested

We evaluated a number of different configurations of our pipeline from Lean to executables, which includes erasure to $\lambda_{\square}$, conversion to Malfunction by the `lbox` tool, and compilation of Malfunction using the OCaml compiler backend.

The reference configuration was to use an OCaml toolchain with Flambda optimizations enabled and the `-O2` optimization flag set, with arithmetic axioms tagged for inlining, constructor pruning and unboxing enabled, and persistent arrays implemented using Baker's trick. We also tested a number of changes to this configuration:

- varying the optimization level applied by the OCaml compiler
  - using the Flambda optimizer with the `-O3` flag
  - using an OCaml toolchain with Flambda but without passing optimization flags ("`-O0`")

[16] https://github.com/inria-cambium/lean-to-lambdabox

  - using an OCaml toolchain without the Flambda optimizer
- using an `axioms.ml` file without inlining annotations on arithmetic axioms
- disabling constructor pruning in erasure, unboxing in `lbox`, or both
- using the `Sek` library or unsafe dynamic arrays to implement Lean's `Array` API.

In every configuration, the Malfunction compiler used to compile the benchmark executables uses OCaml version 5.3.0. These total 10 configurations of the Malfunction pipeline were compared to the "new" Lean compiler, at version 4.22.0, which is the first release version of Lean where the compiler is implemented in Lean.

5.3. Benchmark programs tested

There are few existing Lean benchmarks suitable for measuring the performance of the Malfunction pipeline in its various configurations; the Lean repository features a benchmark suite which is run as part of CI, but this is aimed at measuring Lean's performance as a proof assistant, rather than its performance as a programming language. The paper "Counting Immutable Beans" by Ullrich and de Moura (2021), presenting the Lean compiler's reference counting optimizations with a focus on the speed of compiled programs, does include a number of self-contained benchmarks, including common algorithms such as red-black trees, quicksort, and union-find. These are adequate for our purposes, with only minor modifications to remove features unsupported by the erasure pipeline and convert them to functions of type `Nat -> Nat`. To these benchmarks, we add a few elementary test programs, used to test that the erasure pipeline functions correctly and investigate the performance characteristics of different implementations of some simple functions. In total, we arrive at a set of 10 distinct benchmarks, with variants bringing the total to 19. A detailed description of each benchmark is included in Appendix B.

We selected the parameter n used to run each benchmark such as to obtain an execution time around 0.1s to 1s using the reference configuration of the Malfunction backend.

5.4. Performance results

The benchmark results discussed in this section are shown in Appendix C.1, with full benchmark results in Appendix C.2. The main factor determining the performance of the benchmark executables produced by the Malfunction pipeline is the optimization level used by the OCaml compiler, as seen in Table 1. Unsurprisingly, using an OCaml toolchain without the Flambda optimizer or using Flambda without requesting optimization generates significantly slower code. The benchmark `triangle_rec` is an outlier in this regard: at runtime, it produces a particularly deep call stack, and in OCaml configurations without optimization, the generated code still includes allocations for local function definitions, and hence the garbage collector runs repeatedly and must scan the entire call stack, leading to a dramatic slowdown. Flambda's `-O2` and `-O3` optimization levels prove crucial in order to get good performance from extracted $\lambda_{\square}$ code, which is littered with unused arguments and $\beta\eta$-redexes left over from erasure. The `-O3` level does not yield a large improvement over `-O2`. Table 1 also shows the effect of disabling the forced inlining of arithmetic functions on natural numbers and integers which are provided in the `axioms.ml` file. This mainly provides a large speedup for the benchmarks `even` and `iflazy`, which perform repeated arithmetic operations without any memory allocation and hence benefit the most from eliminating the function call overhead for e.g. decrementing a number. Finally, we can compare the reference configuration of the Malfunction pipeline with Lean's compiler. The Lean compiler generally produces much faster executables from the benchmark programs than the extraction pipeline, with a geometric-mean speedup factor of 2.5 and large variation between the individual benchmarks.[17]

[17] The Malfunction pipeline outperforms the Lean compiler on the `even` benchmark because the latter does not perform tail-call optimization for mutually recursive functions.

The effects of constructor pruning and unboxing are more modest. The benchmark `qsort_fin`, which performs quicksort on arrays of values of type `Fin 2^32`, was expected to show improved performance when these optimizations are activated in the erasure step resp. the `lbox` tool, since pruning and unboxing turns the bounded number type `Fin (2^32)` into just a `Nat` with no indirection. Indeed, the data shown in Table 2 confirm this expectation: the execution time of `qsort_fin` is significantly larger than that of `qsort` when pruning and unboxing are disabled, a gap which largely disappears in the `reference` backend which has these enabled. Interestingly, the `qsort` benchmark itself, which manipulates numbers of type `Nat` which cannot be further affected by pruning and unboxing, also benefits from these optimizations. Inspecting the source code of Lean's `Array.qsort` function, we find that the auxiliary function `qpartition` contains a subset type carrying proof data along with a value in its return type; enabling pruning and unboxing removes this dummy information, explaining the performance gain. Pruning and unboxing also improve the performance of seemingly unrelated benchmarks such as `iflazy`, `binarytrees` and `unionfind`. This is due to Lean's widespread use of `Decidable P` (which becomes equivalent to `Bool` after pruning) to support using propositions as conditions in if-then-else expressions, as well as a number of commonly-used typeclasses, e.g. the `HSub` typeclass for heterogeneous subtraction, whose single constructor introduces an indirection which is removed by unboxing.

The `rbmap` group of benchmarks, which test Lean's implementation of red-black trees used as maps from `Nat` to `Bool`, is interesting to consider closely because the differences between the source code of `rbmap_std`, `rbmap_mono` and `rbmap_raw` are respectively manually performed specialization of polymorphic functions to specific type arguments and removal of proof data. The latter is guaranteed by the Lean compiler's memory representation and handled by pruning and unboxing in the Malfunction pipeline, and it would be possible for the Lean and OCaml compiler to handle the former if their optimizations were sufficiently powerful, in which case we would expect the performance of the three `rbmap` benchmarks mentioned to be identical after compilation. However, the results shown in Table 2 show that this is not quite the case: while `rbmap_mono` and `rbmap_raw` have very close execution times, indicating that propositional content is successfully removed, the Lean compiler produces slightly faster code for `rbmap_mono` than for `rbmap_std`. Inspecting the `C` output produced by the Lean compiler reveals that the `rbmap_std` executable does correctly specialize the various functions on red-black trees to use the natural comparison function on `Nat` but stores values of type `Bool` behind a pointer, as would be necessary for generic values of type unknown at compile-time. The Malfunction backend does not have such a large performance difference between `rbmap_std` and `rbmap_mono`, and we assume that this is due to the OCaml compiler recognizing more opportunities to use unboxed booleans. Rather curiously, the `rbmap_raw` benchmark runs faster than `rbmap_beans` when compiled by Lean but slower when compiled using the Malfunction pipeline. We attribute this to the different optimizers' heuristics performing differently on these specific examples.

Finally, we turn to comparing the three implementations of the axioms required for Lean's `Array α` datatype: the reference configuration using persistent arrays with difference lists (Baker's trick), a tree data structure provided by the `Sek` library, and an unsafe implementation which is not persistent. The results of the relevant benchmarks are shown in Table 3, and Figure 2 shows the scaling of the `qsort_single` benchmark, which sorts a single array, with increasing array size. Using an unsafe, non-persistent implementation for `Array` improves performance by a factor of approximately 3 compared to the reference configuration, and the `Sek` data structure performs worse by a factor of 6 on this quicksort task.[18] Due to the large branching factor used by `Sek`, the additional $\log n$ factor in the asymptotic complexity of operations on trees is only faintly discernible. The benchmark `list_sum_foldr` is also

[18]When considering operations other than those provided by mutable dynamic arrays, however, tree-based sequence data structures show more benefits; for example, the `Sek` library provides logarithmic-time concatenation, whereas concatenation is linear-time with dynamic arrays.

included in Table 3, as it is affected by the choice of array implementation; this is due to the fact that Lean's implementation of `List.foldr` first converts the list to an array before traversing it in reverse order. Finally, `unionfind` also uses array operations, but these do not constitute a majority of its execution time with our Malfunction backends. The `unionfind` benchmark is implemented using a monadic programming style, utilizing Lean's powerful `do`-notation (Ullrich and Moura 2022) to mimic imperative programming with state and exceptions. Programs written using monad transformers employ multiple layers of abstraction to express what would be simple operations in imperative programming languages, and it is essential to the performance of code generated by Lean's compiler that these abstractions are removed by compilation. This is ensured by tagging key functions in Lean with the `@[inline]` attribute. Although the OCaml compiler does support inlining annotations, the $\lambda_{\square}$ and Malfunction languages have no such concept, and hence our code generation pipeline pays the full price of abstraction, performing over 10 times slower on `unionfind` than the Lean compiler. Although `unionfind` is the only example in our benchmark set which exhibits such a large slowdown, we conjecture that it is the benchmark most representative of real-world Lean programs, such as the Lean compiler itself, which heavily use monads.

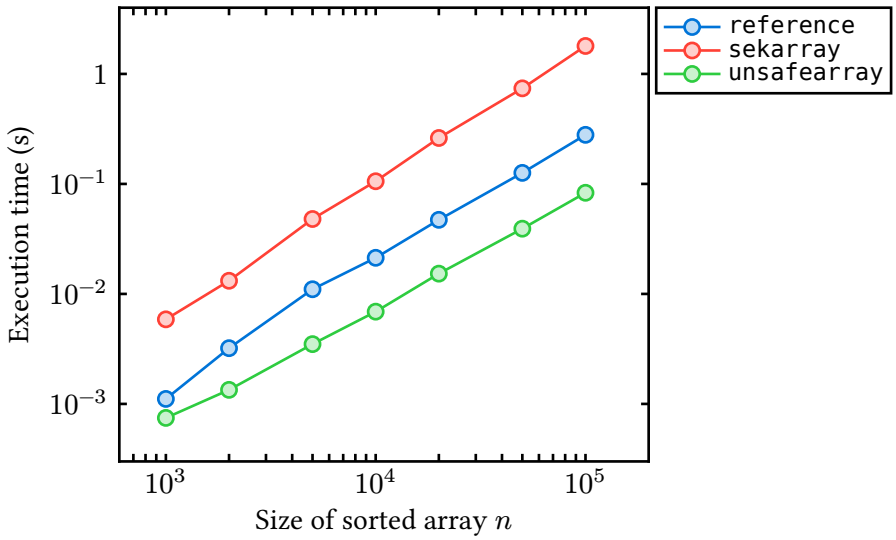

Figure 2: Scaling of the `qsort_single` benchmark with different implementations of `Array`.

6. Discussion

6.1. Related work

As a bridge from one to the other, this work naturally builds upon the existing literature and software concerning the Lean 4 programming language and its compiler, as well as the Rocq proof assistant and extraction therefrom.

Extraction from Rocq (née Coq) was first implemented by Paulin-Mohring (Paulin-Mohring 1989) before being replaced by Letouzey's extraction mechanism (Letouzey 2004), which is still in use today. The modern developments in this area include the verified extraction of Rocq code to Malfunction (Forster, Sozeau, and Tabareau 2024), which uses the erasure procedure to $\lambda_{\square}$ formalized as part of the MetaRocq project (Sozeau et al. 2025). The $\lambda_{\square}$ language is also used in the CertiCoq verified compiler from Rocq to C (Anand et al. 2017), and it is the focus of recent work by Spitters et al. on using $\lambda_{\square}$ and a typed variant $\lambda_{\square}^{T}$ as a point of divergence for extraction from Rocq to a variety of functional languages (Annenkov et al. 2022).

The Lean proof assistant's maturation into a general-purpose programming language was accomplished by the redesign leading to the Lean 4 system (De Moura and Ullrich 2021). The specific

reference counting optimizations performed by Lean's compiler, described by Ullrich and de Moura (2021), form the basis for the Perceus algorithm for reference free garbage counting, as implemented in the Koka language (Reinking et al. 2021). Using a reference counting backend, these techniques allow a style of functional programming known as "functional but in place" and "fully in place", in which the compiler produces functions which modify data structures in-place without requiring additional memory allocation so long as the data structure is unshared (Lorenzen, Leijen, and Swierstra 2023).

Other than Rocq and Lean, a number of other systems inhabit the continuum between dependently typed programming languages and proof assistants, including the Agda proof assistant (Norell 2007), which supports executable generation via extraction to GHC Haskell and JavaScript, the F* programming language (Swamy et al. 2016), intended for program verification, and the Idris programming language (Brady 2021), which aims to provide a fully-fledged language in which to explore the use of expressive type systems for programming. The Isabelle proof assistant, which is based on higher-order logic instead of type theory, features a verified translation of functions into CakeML (Kumar et al. 2014), a formalized subset of Standard ML with a formally verified compiler to assembly language.

6.2. Limitations

The erasure procedure from Lean to $\lambda_{\square}$ implemented during this internship covers the core features of the language, but lacks support for a number of important features. Most centrally, the various functions available in Lean's `IO` monad are not available, which requires the parts of programs performing side-effects to be implemented in OCaml and linked with the program produced by erasure after compilation. This is not a conceptual limitation, and support for IO operations conforming to Lean's `FS.Stream` API and `IO` monad could be added to the erasure pipeline via the `axioms.ml` file. Similarly, Lean's `Task` API for asynchronous computations is currently unsupported, but it could be added using axioms based on an existing OCaml library such as `domainslib` to support parallelism and concurrency.

The benchmark suite used to evaluate the performance of the erasure pipeline is quite limited, with only a handful of simple programs. A more extensive set of benchmarks would provide more data useful for understanding the large variation in benchmark performance between the Lean compiler and our erasure. More importantly, the benchmark suite used here does not feature many programs which manipulate proof data alongside computational data, which is the raison d'être of the erasure step and one of the main features provided by writing programs in a proof assistant, or use a monadic programming style, which is a common building block in more advanced Lean programs. Being able to extract and run the code for the Lean compiler itself, the foremost example of large, complicated software written in Lean, is quite far out of reach given the current state of the erasure mechanism, as the Lean compiler's implementation relies on essentially all esoteric features of the language and interfaces heavily with parts of the code written in C.

As shown by Mytkcowicz, Diwan, Hauswirth and Sweeny (Mytkowicz et al. 2009), work measuring the performance of executables is vulnerable to measurement bias. While we only performed measurements one one machine and made no effort to control for effects due to UNIX environment size or link order, the magnitude by which the Lean compiler outperforms our erasure pipeline in our measurements is large enough that we are confident that this a real effect. However, some of the differences we observe between the various configurations of the erasure pipeline are of a magnitude on the order of a few percentage points or visible in only a few benchmark programs, and more controlled measurements would be necessary in order to confirm these results.

6.3. Conclusion

Dependently typed proof assistants provide a unique environment for writing verified programs, thanks to the dual purpose of the term language as a system for formal mathematical reasoning and

as a language for specifying computations. In this internship, we surveyed the existing mechanisms for producing executable code from the term languages of the Lean and Rocq proof assistants. Although similar in purpose, Lean's compiler and Rocq's verified extraction differ significantly, both in their high-level approach regarding the responsibility for optimization and in implementation details concerning memory management. Our implementation of type erasure from Lean's term language to $\lambda_{\square}$ provides a bridge into Rocq's verified extraction pipeline, enabling the use of the Malfunction programming language and OCaml compiler as a backend for compilation of Lean programs. While lacking a number of features treated specially by the Lean compiler, our erasure procedure supports the core features of Lean's term language into which a variety of conveniences provided by Lean's user-visible syntax and elaborator are desugared. Nontrivial pure programs written in idomatic Lean can thus be processed via our erasure pipeline, including a union-find data structure and the implementations of Quicksort and red-black trees from the Lean standard library. We measured the execution time of the executables produced from a small set of source programs via erasure, with various configurations of the pipeline, in order to gain understanding of the factors influencing performance. Our pipeline performs particularly poorly when applied to code making use of monads to direct computations, mainly because we do not pass on the inlining annotations present in the Lean source code, and execution times are generally on the order of 2.5 times longer than when using the Lean compiler, with large variations between the individual benchmarks. We conjecture that the Lean compiler's middle-end optimization passes are of comparable quality to those run by the OCaml compiler, and that the large performance gap we observe is due to Lean's sophisticated use of reference counting for memory management.

The main weakness of reference counting, cyclic data structures, is absent in Lean and other pure programming languages, as mutability is required to create cyclic references during a program's execution. Therefore, reference counting with Lean-like reuse optimizations appears as an attractive approach with the potential to improve the performance of code extracted from other proof assistants. Valuable future work in this direction could include implementing a new backend for code generation from $\lambda_{\square}$ targeting an intermediate stage of Lean or Koka's compilation, which would enable programs written in Rocq or Agda to benefit from the techniques that make Lean fast. Alternatively, Pinto and Leijen's prototype of OCaml with reference counting (Pinto and Leijen 2023), if extended to include reuse optimizations, could be used to compile programs produced by Rocq's existing extraction mechanism and potentially improve performance. Finally, improving the erasure of Lean to $\lambda_{\square}$, be it by adding support for more builtin language features, more aggressive removal of unnecessary arguments, or preserving more type information and targeting the variant $\lambda_{\square}^{T}$, would enable compiling Lean code with diverse language backends, providing increased interoperability with existing codebases in applications such as smart contracts or systems programming, as well as significant reduction of the trusted computing base when using a verified compiler as the backend.

Acknowledgements

I would like to thank Yannick Forster for the mentorship, encouragement, and advice he gave me during this internship, as well as his feedback on drafts of this report. I am grateful to Arthur Adjedj, Cameron Zwarich and Florian Angeletti for kindly and patiently answering my stream of questions respectively concerning the Lean language, its compiler, and all things OCaml. I thank Gabriel Scherer for convincing me that dynamic persistent arrays were easy to implement and for dedicating half an hour of his time to pair-coding in order to make that true. Finally, I thank the permanent members and my fellow interns on the Cambium team for their warm welcome and diverting discussions over coffee and tea, and especially Poyraz, who provided much of the aforementioned tea.

A: Proofs and programs in dependent type theory

We introduce the term language of dependently typed proof assistants and illustrate its use in writing programs, proofs, and programs containing proofs. For consistency, the code examples are written in a simple fragment of Lean syntax, but the concepts demonstrated apply to all proof assistants based on type theory, as the term languages used are all variants of dependent type theory with inductive types, with only minor differences.

A.1: Functional programming with simple types

At their core, type-theoretic proof assistants are functional programming languages with a strict type system. As a first example, we define the types of natural numbers, which are either zero or the successor of some natural number as in Peano arithmetic, and booleans, and then define a function `div2` which computes the division by two and remainder of any natural number.

```
inductive Nat where
| zero: Nat
| succ (n: Nat): Nat

inductive Bool where
| false: Bool
| true: Bool

def div2 (n: Nat): Nat × Bool :=
  match n with
  | .zero => (.zero, .true)
  | .succ m =>
    let (k, b) := div2 m;
    match b with
    | .true => (.succ k, .false)
    | .false => (k, .true)
```

Here the types `Nat` and `Bool` are defined as inductive datatypes, which correspond to algebraic datatypes in ML-family languages. It is also possible to define a group of mutually inductive datatypes together. Pattern matching – the `match` construct – is used to inspect a value of an inductive type and obtain the constructor and constructor arguments used to create it, returning a different value in each case. The function `div2` uses recursion to perform the computation, calling itself on smaller arguments and terminating when the base case is reached.

Proof assistants are functional programming languages, in which functions are values like any other and may be passed as arguments to, and returned from, functions. In the preceding example, a top-level definition gives the name `div2` to a value with type `Nat -> Nat × Bool`. Functions can also be created locally, without naming them, by lambda-abstraction using the `fun` keyword.

```
def some_value: Bool × (Nat -> Nat) := (.true, fun n: Nat => n)
```

Functions which take multiple arguments are most often represented by using chained implication, commonly known as currying. For example, the addition function on natural numbers, named `Nat.add`, has type `Nat -> (Nat -> Nat)`, most often written `Nat -> Nat -> Nat`, with the convention that the "function type" arrow is right-associative. Function application is written by juxtaposition and implicitly left-associative: `Nat.add 3` is the function which to a given number adds 3, and `Nat.add 3 5` is 8. For legiblity, the notation `3 + 5` is also available.

A.2: Dependent types

One major difference between dependently typed proof assistants and conventional functional languages is that types are also first-class: types are themselves values and may be named, passed to and

returned from functions. These values also have a type, which is called `Type`.[19] For example, we can define a function `boolvec: Nat -> Type` such that `boolvec n` corresponds to the type of sequences of exactly `n` boolean values:

```
def boolvec (n: Nat): Type :=
  match n with
  | .zero => Unit
  | .succ m => Bool × boolvec m

def b10: Bool × (Bool × Unit) := (.true, (.false, ()))
def b110: boolvec 3 := (.true, b10)
```

Here it is correct to write the type of `b10` as either `Bool × (Bool × Unit)` or `boolvec 2`, because the proof assistant's type checker can see the definition of `boolvec` and determine that the two are equal. However, the type system would forbid the ill-typed `def b110: boolvec 2 := (.true, b10)`, because the types `boolvec 2` and `boolvec 3` are different! This is an instance of dependent typing, where a type (here, `boolvec n`) depends on a value (here, `n`). In fact, the type system can express reasoning about `boolvec n` even when `n` is not a fixed constant:

```
def alltrue (n: Nat): boolvec n :=
  match n with
  | .zero => ()
  | .succ m => (.true, alltrue m)

def double (n: Nat) (v: boolvec n): boolvec (2*n) :=
  match n with
  | .zero => ()
  | .succ m =>
    let (h, t) := v;
    (h, (h, double m t))
```

The function `alltrue` has a so-called dependent function type, written `(n: Nat) -> boolvec n`, which specifies that the result of `alltrue n` has type `boolvec n`, whatever the value of `n`. In order to check that the return value has the appropriate type, the type checker verifies that, since `alltrue m` is of type `boolvec m`, the value `(.true, alltrue m)` is of type `boolvec (.succ m)`, which matches `boolvec n` because `n` and `.succ m` are equal in this branch of the `match` expression. Likewise, the type `double: (n: Nat) -> boolvec n -> boolvec (2*n)` encodes within the type system that applying the function to a boolvector doubles its length. In fact, regular function types are a special case of dependent function types where the codomain type does not depend on the value of the argument.

Since types are also values with their own type `Type`, dependent function types can also be used to write functions which take a type as an argument, which is how polymorphic functions are defined:

```
def id (α: Type): α -> α := fun a: α => a

def id_bool: Bool -> Bool := id Bool
def still_one := id Nat (.succ zero)
```

The polymorphic identity function `id` itself has the type `(α: Type) -> α -> α`. This can be read as "for any type `α`, `id α` has type `α -> α`", and indeed, `id Bool` has type `Bool -> Bool` and `id Nat` has type `Nat -> Nat`. The argument `α` is not optional: the expression `id 3` is ill-typed, because `3` is not a type. This is a difference from ML-style type systems, where instantiations of polymorphic functions at specific types are not distinguished at the term level.

[19] For reasons related to the logical consistency of the system, `Type` is not quite of type `Type`, but assuming `Type : Type` is an adequate simplification for the purposes of this report.

Just as dependent function types generalize regular function types, pair types can be generalized into dependent pair types, where the type of the second element in a pair may depend on the value of the first. For example, we may define a type `boollist`, representing a list of booleans of any length as a pair containing some integer `n` and a `boolvec` of length exactly `n`:

```
def boollist := (n: Nat) × (boolvec n)

def list_double (l: boollist): boollist :=
  let ⟨(n: Nat), (v: boolvec n)⟩ := l;
  ⟨2*n, double n v⟩
```

The input and output of `list_double` carry two pieces of information, both the length of the result vector and the boolvector itself.

Using dependent functions and pairs, programs written in dependently typed languages can express detailed information about their behavior directly at the type level.

A.3: Logic using types

Beyond their use in programming, the dependent type systems used in proof assistants are rich enough to encode sophisticated logical reasoning via the Curry-Howard correspondence, in which logical propositions are considered as types. For any proposition `A`, a term `a` of type `A` represents evidence that the proposition `A` holds. This is written `A: Prop` (read: "`A` is a proposition") and `a: A` ("`a` is evidence for `A`"). To prove a proposition, then, is to provide evidence by writing a term which has the appropriate type; the type-correctness of the term corresponds to a logical derivation of the proposition.

Using this correspondence, the proposition `True` is the type with a single element `True.intro`, and the proposition `False` is an empty type with no values. The usual logical connectives have simple representations:

- Conjunction is represented by the type `A ∧ B` of pairs `⟨a,b⟩` with `a: A` and `b: B`
- Disjunction is represented by an inductive type `A ∨ B` whose values can be introduced with two constructors, either `.inl a` for some `a: A` or `.inl b` for `b: B`
- Implication is represented by the type of functions `A -> B`.

These definitions exactly follow the Brouwer–Heyting–Kolmogorov interpretation of the notion of proofs in intuitionistic logic.

Proofs are built using the language features introduced above, which work the same for logical content as for computational data. For example, consider the tautology that for any propositions A, B and C, supposing that $A \to C$ and $B \to C$ hold we also have $A \vee B \to C$. The proof term for this statement, whose type is `(A -> C) -> (B -> C) -> (A ∨ B) -> C`, closely follows the structure of the natural-language formal proof:

```
def tauto_proof (ha: A -> C) (hb: B -> C): (A ∨ B) -> C :=
  fun h : A ∨ B =>
    match h with
    | .inl a => ha a
    | .inr b => hb b
```

Suppose $A \vee B$ holds. The evidence for $A \vee B$ must be either evidence for A or evidence for B. If A holds, then by the first hypothesis, C holds; if B holds, then by the second hypothesis, C holds. So we know that C holds in both cases. Therefore, $A \vee B$ implies C.

Propositions may also depend on terms of computationally relevant types: predicates over data of a certain type are represented as functions with codomain `Prop`, the type of propositions. For instance, the `nonzero` predicate over natural numbers could be defined as

```
def nonzero (n: Nat): Prop :=
  match n with
  | .zero => False
  | .succ _ => True
```

Universally quantified propositions are represented using dependent function types. For instance, the statement "all natural numbers are nonzero" is represented by the proposition `(n: Nat) -> nonzero n`. In Lean, reading dependent function types as universal quantification is further encouraged by the equivalent alternative syntax `∀ n: Nat, nonzero n`, but for consistency we will use only the arrow notation. Note that, since this particular statement is false, it is possible to define it as a proposition, but not to prove it. Indeed, the required evidence – a function mapping every `n` to evidence of `nonzero n` – is impossible to construct, since it would have to return a value in `False`, which is an empty type, when `n` is taken to be `0`.

One common way to prove universal statements about the natural numbers is proof by induction, in which a statement of the form `(n: Nat) -> P n` is proven from a base case proving `P 0` and an induction case proving `(n: Nat) -> P n -> P (.succ n)`. In type-theoretic proof assistants, this corresponds to defining a proof term by structural recursion, similarly to the functions `div2` or `ones` defined above. In greater generality, computation by recursion and proof by induction can be used with any inductive type. Unrestricted recursion, with potentially nonterminating functions, would allow proofs of `True -> False` by constructing a function which loops forever without ever returning a value. To prevent this source of inconsistency, proof assistants employ various termination checking methods in order to ensure that the type checker only accepts total functions.

Finally, like dependent function types, dependent pair types also have a logical interpretation. In fact, there are two variants of dependent pair types involving logical content:

- The existentially quantified proposition `∃ x: X, P x`, which represents the *proposition* "there exists an `x` such that `P x` holds"
- The subtype `{ x: X // P x }`, which is the *type* of all `x` such that `P x` holds.

In both cases, a value is constructed by providing some `x` of type `X` together with evidence for the proposition `P x`. The fact that `∃ x: X, P x` has type `Prop` while `{ x: X // P x }` has type `Type` is crucial, because the type system ensures a separation between `Prop` and `Type`: evidence of propositions in `Prop` may not impact the computation of values of types in `Type`. For example, given some nonzero natural number `k: { n: Nat // nonzero n }` it is possible to know "which number it is", whereas no specific number may be retrieved from evidence of the proposition `∃ n: Nat, nonzero n`.

The features of the term language demonstrated so far are the building blocks upon which all proofs in type-theoretic proof assistants are built, and they are sufficient to express definitions and proofs across modern mathematics, as witnessed by the various libraries of formalized results.

A.4: Programming with proofs

Since the term language in type-theoretic proof assistants can express both computations and logic, it can be used to write programs which mix computational code together with logical parts expressing invariants, well-formedness of data, and specifications of programs, in order to obtain formally verified programs. For example, the following toy program bundles together the computation of a^k, where a and k are natural numbers and a is assumed to be nonzero, with the proof that the result is also nonzero:

```
def pow (a: Nat) (ha: a > 0) (k: Nat): { n : Nat // n > 0 } :=
  match k with
  | .zero => ⟨1, Nat.one_pos⟩
  | .succ k' =>
    let ((m: Nat), (hm: m > 0)) := pow a ha k';
    let n: Nat := a * m;
```

```
    let hn: n > 0 := @Nat.mul_pos a m ha hm;
    ⟨n, hn⟩
```

Here the recursive call returns a dependent pair which is destructured in order to obtain both the computational value of a^{k-1} and the proof that this value is nonzero. Notably, data and proofs are treated identically in the arguments and body of `pow`. The use of the lemmas `Nat.one_pos: 1 > 0` and `@Nat.mul_pos: (a: Nat) -> (b: Nat) -> (a > 0) -> (b > 0) -> (a*b > 0)` , which are admittedly relatively trivial in this case, illustrates the reuse of mathematical facts proven in preexisting libraries.

Since the type checker accepts the definition of `pow`, it is certain that the implementation is correct with respect to the specification given by its type.

B: Detailed presentation of the benchmarks

B.1: Simple benchmarks

A first group of benchmarks comprises simple programs whose runtime is linear in n:

- `even` computes the parity of n using two mutually recursive functions `even` and `odd`: $n + 1$ is even when n is odd and vice versa, terminating at 0.
- `iflazy` is defined as `def iflazy (n: Nat): Nat := if n = 0 then 42 else iflazy (n-1)` and serves as a test of the fact that the `if` notation is correctly converted into a branching construct instead of always evaluating both alternatives.
- The `list_sum` benchmarks create a list of size n full of ones and then compute the sum of all entries. Three different implementations are used for the summation step:
  - `list_sum_foldl` performs a left fold with addition using the function `List.foldl`
  - `list_sum_foldr` performs a right fold, using `List.foldr` with addition. This corresponds to the behavior of `List.sum` in Lean.
  - `list_sum_rev` first reverses the list and then performs a left fold, which is the traditional way of performing a right fold with only tail-recursive functions.
- The `triangle` benchmarks compute the triangular numbers $0 + 1 + ... + (n - 1)$. Here we also compare three implementations:
  - `triangle_acc` uses a tail-recursive auxiliary function with an accumulator argument.
  - `triangle_foldl` creates the list `[0, …, n-1]` using `List.range` and then performs a left fold.
  - `triangle_rec` naïvely applies the identity `triangle (n+1) = n + triangle n`, using linear stack space as the function is not tail-recursive.

B.2: Benchmarks from Counting Beans

These benchmarks were adapted from the Counting Beans paper (Ullrich and de Moura 2021) by removing all I/O code appearing within the benchmarked functions, replacing instances of fixed-width integer types with `Nat`, and replacing uses of Lean's `Task` primitive for asynchronous computation with synchronous operations.

One group of benchmarks consists of programs manipulating tree-like data structures, which are ubiquitous in functional programs and notably include symbolic representations of expressions, the main data manipulated in the implementations of compilers and proof assistants.

- `binarytrees` constructs many complete binary trees of depths up to n and traverses them.
- `const_fold` performs constant folding on depth-n symbolic natural number arithmetic expressions
- `deriv` computes symbolically the nth derivative of the function $x \mapsto x^x$
- The `rbmap` benchmarks generate a red-black tree with n nodes, used as a map from `Nat` to `Bool`, and fold a simple function over it. We compare four variants:

- `rbmap_std` uses an implementation of `RBMap` copied verbatim from Lean's standard library. The implementation is polymorphic over any key and value type, and evidence for a well-formedness predicate is bundled together with the tree.
- `rbmap_mono` specializes the standard-library implementation to monomorphically use `Nat` as the type of keys and `Bool` as the type of values. The well-formedness evidence is still present.
- `rbmap_raw` additionally removes the well-formedness evidence from the specialized data structure.
- `rbmap_beans` uses the `RBMap` implementation from the Counting Beans paper (Ullrich and de Moura 2021), which is specialized to `Nat` and `Bool` and dispenses with evidence for well-formedness just like `rbmap_raw`, but uses a slightly different rebalancing algorithm.

The last group of benchmarks tests more elaborate code involving arrays and monads. These will show the cost of the different ways to implement persistent arrays.

- The three `qsort` benchmarks generate pseudorandom arrays and use Lean's existing `Array.qsort` function to sort them.
  - `qsort` sorts arrays of `Nat` with increasing sizes from 0 to $n - 1$.
  - `qsort_fin` sorts arrays of `Fin (2^32)`, with sizes increasing like `qsort`.
  - `qsort_single` sorts a single array of `Nat` with size n.
- `unionfind` implements a union-find data structure on n nodes using a backing array, performs a sequence of union operations, and counts the number of nodes which are not the representative of their set. This benchmark is particularly interesting as it uses monads, both to present the union-find structure as a mutable object and to allow for exceptions.

C: Tables of benchmark results

C.1: Benchmarks mentioned in the main text

benchmark \ backend	`lean`	`noflambda`	`flambda-O0`	`reference (-O2)`	`flambda-O3`	`noinline`
`even`	2.26	1.26	2.11	1.00	0.99	2.27
`iflazy`	0.14	2.58	8.90	1.00	1.00	2.66
`list_sum` (group)	0.30	1.21	1.71	1.00	1.00	1.13
`triangle` (group)	0.50	5.17	15.68	1.00	0.99	1.41
`binarytrees`	0.79	2.37	8.12	1.00	1.00	1.46
`const_fold`	0.26	1.40	3.96	1.00	1.00	1.02
`deriv`	0.43	1.31	2.23	1.00	0.99	1.02
`rbmap` (group)	1.16	1.21	1.68	1.00	1.02	1.01
`qsort` (group)	0.25	1.66	3.70	1.00	0.86	1.02
`unionfind`	0.08	1.18	2.32	1.00	0.99	1.01
Overall (geom. mean)	0.40	1.71	3.73	1.00	0.98	1.31

Table 1: Effect of Malfunction backend optimization settings on benchmark execution time, adjusted for startup time and relative to the reference configuration. Lower values indicate faster execution. For benchmarks with multiple variants, the value reported is the geometric mean of all variants.

benchmark \ backend	lean	reference	prune-nounbox	noprune-unbox	noprune-nounbox
iflazy	65 ms (0.14)	482 ms (1.00)	709 ms (1.47)	625 ms (1.30)	839 ms (1.74)
binarytrees	433 ms (0.79)	551 ms (1.00)	620 ms (1.13)	609 ms (1.11)	617 ms (1.12)
qsort	160 ms (0.31)	517 ms (1.00)	553 ms (1.07)	562 ms (1.09)	589 ms (1.14)
qsort_fin	162 ms (0.31)	524 ms (1.00)	622 ms (1.19)	607 ms (1.16)	653 ms (1.25)
rbmap_std	838 ms (1.18)	711 ms (1.00)	715 ms (1.01)	722 ms (1.02)	722 ms (1.02)
rbmap_mono	772 ms (1.11)	697 ms (1.00)	716 ms (1.03)	717 ms (1.03)	717 ms (1.03)
rbmap_raw	762 ms (1.09)	697 ms (1.00)	712 ms (1.02)	716 ms (1.03)	717 ms (1.03)
rbmap_beans	824 ms (1.26)	653 ms (1.00)	650 ms (0.99)	657 ms (1.01)	659 ms (1.01)
unionfind	79 ms (0.08)	926 ms (1.00)	1042 ms (1.12)	930 ms (1.00)	1047 ms (1.13)

Table 2: Effect of constructor pruning and unboxing on the execution time of selected benchmarks. Values are adjusted for executable startup time.

benchmark \ backend	lean	reference	sekarray	unsafearray
list_sum_foldr	0.25	1.00	3.86	1.01
qsort	0.31	1.00	4.66	0.50
qsort_fin	0.31	1.00	4.64	0.49
qsort_single	0.16	1.00	6.43	0.30
unionfind	0.08	1.00	1.58	0.94

Table 3: Effect of different implementations of `Array` axioms on the execution time of selected benchmarks. Values are adjusted for executable startup time and relative to the reference configuration.

C.2: Full results

The first entry is the median execution time of the large run, with the median runtime of the $n = 0$ run subtracted. The second entry is the standard deviation in the large run's execution time; deviations in the calibration run's execution time were on the order of 0.1 ms. The third entry is the runtime relative to the reference configuration.

benchmark \ backend	lean	reference	noflambda	flambda-O0	flambda-O3	noinline
even	130 ms ± 1 ms (2.26)	57 ms ± 0 ms (1.00)	72 ms ± 1 ms (1.26)	121 ms ± 0 ms (2.11)	57 ms ± 0 ms (0.99)	130 ms ± 1 ms (2.27)

benchmark \ backend	lean	reference	noflambda	flambda-O0	flambda-O3	noinline
iflazy	65 ms ± 0 ms (0.14)	482 ms ± 1 ms (1.00)	1243 ms ± 1 ms (2.58)	4289 ms ± 3 ms (8.90)	483 ms ± 4 ms (1.00)	1282 ms ± 2 ms (2.66)
list_sum_foldl	194 ms ± 0 ms (0.33)	590 ms ± 2 ms (1.00)	697 ms ± 2 ms (1.18)	889 ms ± 2 ms (1.50)	579 ms ± 1 ms (0.98)	655 ms ± 2 ms (1.11)
list_sum_foldr	261 ms ± 0 ms (0.25)	1038 ms ± 3 ms (1.00)	1327 ms ± 2 ms (1.28)	2277 ms ± 3 ms (2.19)	1068 ms ± 4 ms (1.03)	1214 ms ± 3 ms (1.17)
list_sum_rev	194 ms ± 0 ms (0.33)	588 ms ± 3 ms (1.00)	696 ms ± 22 ms (1.18)	888 ms ± 10 ms (1.51)	577 ms ± 1 ms (0.98)	655 ms ± 2 ms (1.11)
triangle_foldl	192 ms ± 0 ms (0.31)	625 ms ± 2 ms (1.00)	685 ms ± 1 ms (1.10)	821 ms ± 1 ms (1.31)	597 ms ± 2 ms (0.96)	686 ms ± 2 ms (1.10)
triangle_acc	26 ms ± 0 ms (0.28)	92 ms ± 0 ms (1.00)	188 ms ± 0 ms (2.05)	427 ms ± 0 ms (4.66)	92 ms ± 0 ms (1.00)	167 ms ± 1 ms (1.82)
triangle_rec	249 ms ± 1 ms (1.45)	172 ms ± 1 ms (1.00)	10532 ms ± 1 ms (61.26)	108193 ms ± 37 ms (629.30)	172 ms ± 1 ms (1.00)	241 ms ± 1 ms (1.40)
rbmap_beans	824 ms ± 1 ms (1.26)	653 ms ± 1 ms (1.00)	741 ms ± 1 ms (1.13)	1066 ms ± 4 ms (1.63)	653 ms ± 4 ms (1.00)	666 ms ± 0 ms (1.02)
rbmap_std	838 ms ± 1 ms (1.18)	711 ms ± 1 ms (1.00)	917 ms ± 1 ms (1.29)	1352 ms ± 1 ms (1.90)	714 ms ± 1 ms (1.00)	673 ms ± 3 ms (0.95)
rbmap_mono	772 ms ± 3 ms (1.11)	697 ms ± 0 ms (1.00)	848 ms ± 4 ms (1.22)	1127 ms ± 5 ms (1.62)	722 ms ± 0 ms (1.04)	728 ms ± 1 ms (1.04)
rbmap_raw	762 ms ± 0 ms (1.09)	697 ms ± 1 ms (1.00)	851 ms ± 1 ms (1.22)	1110 ms ± 2 ms (1.59)	722 ms ± 3 ms (1.04)	728 ms ± 1 ms (1.04)
binarytrees	433 ms ± 1 ms (0.79)	551 ms ± 1 ms (1.00)	1303 ms ± 3 ms (2.37)	4469 ms ± 10 ms (8.12)	552 ms ± 0 ms (1.00)	806 ms ± 3 ms (1.46)
const_fold	209 ms ± 1 ms (0.26)	799 ms ± 1 ms (1.00)	1122 ms ± 28 ms (1.40)	3162 ms ± 8 ms (3.96)	799 ms ± 23 ms (1.00)	815 ms ± 4 ms (1.02)
deriv	150 ms ± 2 ms (0.43)	353 ms ± 1 ms (1.00)	462 ms ± 1 ms (1.31)	788 ms ± 6 ms (2.23)	350 ms ± 1 ms (0.99)	359 ms ± 2 ms (1.02)

benchmark \ backend	lean	reference	noflambda	flambda-O0	flambda-O3	noinline
qsort	160 ms ± 0 ms (0.31)	517 ms ± 0 ms (1.00)	915 ms ± 0 ms (1.77)	2200 ms ± 18 ms (4.26)	435 ms ± 0 ms (0.84)	522 ms ± 1 ms (1.01)
qsort_fin	162 ms ± 0 ms (0.31)	524 ms ± 1 ms (1.00)	1020 ms ± 1 ms (1.95)	2343 ms ± 8 ms (4.47)	447 ms ± 0 ms (0.85)	548 ms ± 0 ms (1.05)
qsort_single	45 ms ± 0 ms (0.16)	279 ms ± 0 ms (1.00)	369 ms ± 0 ms (1.32)	745 ms ± 2 ms (2.67)	251 ms ± 0 ms (0.90)	282 ms ± 0 ms (1.01)
unionfind	79 ms ± 0 ms (0.08)	926 ms ± 3 ms (1.00)	1090 ms ± 2 ms (1.18)	2153 ms ± 9 ms (2.32)	916 ms ± 5 ms (0.99)	940 ms ± 5 ms (1.01)

benchmark \ backend	prune-nounbox	noprune-unbox	noprune-nounbox	sekarray	unsafearray
even	57 ms ± 0 ms (1.00)	58 ms ± 0 ms (1.01)	58 ms ± 0 ms (1.01)	56 ms ± 0 ms (0.98)	59 ms ± 1 ms (1.03)
iflazy	709 ms ± 2 ms (1.47)	625 ms ± 1 ms (1.30)	839 ms ± 6 ms (1.74)	480 ms ± 1 ms (1.00)	650 ms ± 0 ms (1.35)
list_sum_foldl	589 ms ± 2 ms (1.00)	587 ms ± 1 ms (0.99)	586 ms ± 3 ms (0.99)	589 ms ± 3 ms (1.00)	585 ms ± 0 ms (0.99)
list_sum_foldr	1065 ms ± 3 ms (1.03)	1044 ms ± 3 ms (1.01)	1063 ms ± 1 ms (1.02)	4009 ms ± 33 ms (3.86)	1046 ms ± 3 ms (1.01)
list_sum_rev	592 ms ± 2 ms (1.01)	587 ms ± 3 ms (1.00)	586 ms ± 2 ms (1.00)	587 ms ± 1 ms (1.00)	584 ms ± 2 ms (0.99)
triangle_foldl	624 ms ± 3 ms (1.00)	616 ms ± 2 ms (0.99)	616 ms ± 1 ms (0.99)	617 ms ± 2 ms (0.99)	614 ms ± 4 ms (0.98)
triangle_acc	97 ms ± 0 ms (1.06)	92 ms ± 0 ms (1.00)	96 ms ± 0 ms (1.05)	93 ms ± 0 ms (1.01)	90 ms ± 0 ms (0.98)
triangle_rec	176 ms ± 1 ms (1.03)	172 ms ± 1 ms (1.00)	176 ms ± 1 ms (1.02)	171 ms ± 1 ms (1.00)	155 ms ± 1 ms (0.90)
rbmap_beans	650 ms ± 4 ms (0.99)	657 ms ± 0 ms (1.01)	659 ms ± 3 ms (1.01)	662 ms ± 0 ms (1.01)	663 ms ± 1 ms (1.01)

benchmark \ backend	prune-nounbox	noprune-unbox	noprune-nounbox	sekarray	unsafearray
rbmap_std	715 ms ± 1 ms (1.01)	722 ms ± 3 ms (1.02)	722 ms ± 1 ms (1.02)	715 ms ± 1 ms (1.01)	683 ms ± 1 ms (0.96)
rbmap_mono	716 ms ± 1 ms (1.03)	717 ms ± 0 ms (1.03)	717 ms ± 1 ms (1.03)	702 ms ± 1 ms (1.01)	696 ms ± 3 ms (1.00)
rbmap_raw	712 ms ± 2 ms (1.02)	716 ms ± 1 ms (1.03)	717 ms ± 1 ms (1.03)	702 ms ± 1 ms (1.01)	695 ms ± 0 ms (1.00)
binarytrees	620 ms ± 0 ms (1.13)	609 ms ± 2 ms (1.11)	617 ms ± 5 ms (1.12)	551 ms ± 0 ms (1.00)	556 ms ± 1 ms (1.01)
const_fold	802 ms ± 5 ms (1.00)	800 ms ± 12 ms (1.00)	801 ms ± 12 ms (1.00)	798 ms ± 3 ms (1.00)	798 ms ± 22 ms (1.00)
deriv	354 ms ± 2 ms (1.00)	352 ms ± 2 ms (1.00)	359 ms ± 1 ms (1.02)	346 ms ± 1 ms (0.98)	351 ms ± 1 ms (1.00)
qsort	553 ms ± 1 ms (1.07)	562 ms ± 0 ms (1.09)	589 ms ± 0 ms (1.14)	2408 ms ± 6 ms (4.66)	260 ms ± 0 ms (0.50)
qsort_fin	622 ms ± 3 ms (1.19)	607 ms ± 3 ms (1.16)	653 ms ± 2 ms (1.25)	2429 ms ± 12 ms (4.64)	256 ms ± 0 ms (0.49)
qsort_single	279 ms ± 0 ms (1.00)	286 ms ± 0 ms (1.02)	285 ms ± 0 ms (1.02)	1797 ms ± 3 ms (6.43)	83 ms ± 0 ms (0.30)
unionfind	1042 ms ± 6 ms (1.12)	930 ms ± 1 ms (1.00)	1047 ms ± 1 ms (1.13)	1466 ms ± 4 ms (1.58)	871 ms ± 3 ms (0.94)

Bibliography

Aasa, Annika, Sören Holmström, and Christina Nilsson. 1988. "An Efficiency Comparison of Some Representations of Purely Functional Arrays". *BIT Numerical Mathematics* 28 (3): 489–503. https://doi.org/10.1007/BF01941130.

Anand, Abhishek, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary B´elanger, Matthieu Sozeau, and Matthew Weaver. 2017. "CertiCoq: A Verified Compiler for Coq". In *CoqPL'17: The Third International Workshop on Coq for Programming Languages*.

Annenkov, Danil, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. "Extracting Functional Programs from Coq, In Coq". *Journal of Functional Programming* 32 (January). https://doi.org/10.1017/S0956796822000077.

Brady, Edwin. 2021. "Idris 2: Quantitative Type Theory in Practice". In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, edited by Anders Møller and Manu Sridharan, 194:1–

26. Leibniz International Proceedings in Informatics (Lipics). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9.

Conchon, Sylvain, and Jean-Christophe Filliâtre. 2007. "A Persistent Union-Find Data Structure". In *Proceedings of the 2007 Workshop on Workshop on ML*, 37–46. Freiburg Germany: ACM. https://doi.org/10.1145/1292535.1292541.

Dolan, Stephen. 2016. "Malfunctional Programming". In *ML Family Workshop 2016*. https://stedolan.net/talks/2016/malfunction/malfunction.pdf.

Eberl, Manuel, Gerwin Klein, Peter Lammich, Andreas Lochbihler, Tobias Nipkow, Larry Paulson, René Thiemann, and Dmitriy Traytel, eds. 2004. "Archive of Formal Proofs". https://www.isa-afp.org/.

Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. "The Essence of Compiling with Continuations". *SIGPLAN Not.* 28 (6): 237–47. https://doi.org/10.1145/173262.155113.

Forster, Yannick, Matthieu Sozeau, and Nicolas Tabareau. 2024. "Verified Extraction from Coq to OCaml". *Proceedings of the ACM on Programming Languages* 8 (PLDI): 52–75. https://doi.org/10.1145/3656379.

Granlund, Torbjörn, and the GMP development team. 1991. "GNU MP: The GNU Multiple Precision Arithmetic Library". 1991. https://gmplib.org/.

Jung, Ralf, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. "Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic". *Journal of Functional Programming* 28. https://doi.org/10.1017/S0956796818000151.

Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, et al. 2009. "seL4: Formal Verification of an OS Kernel". In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 207–20. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery. https://doi.org/10.1145/1629575.1629596.

Kumar, Ramana, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. "CakeML: A Verified Implementation of ML". In *Principles of Programming Languages (POPL)*, 179–91. ACM Press. https://doi.org/10.1145/2535838.2535841.

Leroy, Xavier. 2009. "Formal Verification of a Realistic Compiler". *Commun. ACM* 52 (7): 107–15. https://doi.org/10.1145/1538788.1538814.

Letouzey, Pierre. 2004. "Programmation Fonctionnelle Certifiée : L'extraction De Programmes Dans L'assistant Coq. (Certified Functional Programming : Program Extraction within the Coq Proof Assistant)". PhD thesis. https://tel.archives-ouvertes.fr/tel-00150912.

Lorenzen, Anton, Daan Leijen, and Wouter Swierstra. 2023. "FP^2: Fully in-Place Functional Programming (TR)". https://www.microsoft.com/en-us/research/publication/fp2-fully-in-place-functional-programming/.

Mahboubi, Assia, and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.7118596.

The mathlib community. 2020. "The Lean Mathematical Library". In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 367–81. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery. https://doi.org/10.1145/3372885.3373824.

Maurer, Luke, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. "Compiling Without Continuations". In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 482–94. Barcelona Spain: ACM. https://doi.org/10.1145/3062341.3062380.

McBride, Conor, and James McKinna. 2004. “Functional Pearl: I Am Not a Number–I Am a Free Variable”. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 1–9. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1017472.1017477.

Miné, Antoine, Xavier Leroy, and Pascal Cuoq. 2011. “The Zarith Library”. 2011. https://github.com/ocaml/Zarith/.

Moura, Leonardo De, and Sebastian Ullrich. 2021. “The Lean 4 Theorem Prover and Programming Language”. Edited by André Platzer and Geoff Sutcliffe. *Automated Deduction – CADE 28*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-79876-5_37.

Mytkowicz, Todd, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. “Producing Wrong Data Without Doing Anything Obviously Wrong!”. *SIGPLAN Not.* 44 (3): 265–76. https://doi.org/10.1145/1508284.1508275.

Nielsen, Eske Hoy, and Bas Spitters. 2025. “lbox Extraction Backend”. 2025. https://github.com/AU-COBRA/lambda-box-extraction.

Nipkow, Tobias, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, eds. 2002. “Isabelle/HOL”. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-45949-9.

Norell, Ulf. 2007. “Towards a Practical Programming Language Based on Dependent Type Theory”. PhD thesis. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

Paulin-Mohring, Christine. 1989. “Extracting F_ω's Programs from Proofs in the Calculus of Constructions”. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*. Austin: ACM. https://doi.org/10.1145/75277.75285.

Peter, David. 2024. “hyperfine”. November 2024. https://github.com/sharkdp/hyperfine.

Pinto, Elton, and Daan Leijen. 2023. “Exploring Perceus for OCaml”. https://www.microsoft.com/en-us/research/publication/exploring-perceus-for-ocaml/.

Pottier, François, and Arthur Charguéraud. 2020. “An Efficient Ocaml Implementation of Ephemeral and Persistent Sequences”. 2020. https://gitlab.inria.fr/fpottier/sek.

Pottier, Loic. 2001. “Extraction Dans Le Calcul Des Constructions Inductives”. In *Journées Francophones Des Langages Applicatifs (JFLA'01), Pontarlier, France, Janvier, 2001*, edited by Pierre Castéran, 49–58. Collection Didactique. INRIA.

Reinking, Alex, Ningning Xie, Leonardo De Moura, and Daan Leijen. 2021. “Perceus: Garbage Free Reference Counting with Reuse”. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. Virtual Canada: ACM. https://doi.org/10.1145/3453483.3454032.

The Rocq Development Team. 2025. “The Rocq Prover”. Zenodo. April 2025. https://doi.org/10.5281/zenodo.15149629.

Shankar, Natarajan. 1999. “Efficiently Executing PVS”. Menlo Park, CA.

Sozeau, Matthieu, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. “Correct and Complete Type Checking and Certified Erasure for Coq, In Coq”. *J. ACM* 72 (1): 1–74. https://doi.org/10.1145/3706056.

Swamy, Nikhil, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, et al. 2016. “Dependent Types and Multi-Monadic Effects in F*”. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, 256–70. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery. https://doi.org/10.1145/2837614.2837655.

Ullrich, Sebastian, and Leonardo de Moura. 2021. "Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming". In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. IFL '19. Singapore, Singapore: Association for Computing Machinery. https://doi.org/10.1145/3412932.3412935.

Ullrich, Sebastian, and Leonardo de Moura. 2022. "'Do' Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl)". *Proc. ACM Program. Lang.* 6 (ICFP). https://doi.org/10.1145/3547640.

Wadler, Philip. 1997. "How to Declare an Imperative". *ACM Comput. Surv.* 29 (3): 240–63. https://doi.org/10.1145/262009.262011.

Yang, Xuejun, Yang Chen, Eric Eide, and John Regehr. 2011. "Finding and Understanding Bugs in C Compilers". In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 283–94. PLDI '11. San Jose, California, USA: Association for Computing Machinery. https://doi.org/10.1145/1993498.1993532.