NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENT OF COMPUTER SCIENCE

TIAGO SILVA MEIRIM

BSc in Computer Science and Engineering

FROM OCAML TO CAKEML, AND BACK

A PIPELINE FOR VERIFIED CODE BY CONSTRUCTION

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

Draft: *July 10, 2025*

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENT OF COMPUTER SCIENCE

FROM OCAML TO CAKEML, AND BACK

A PIPELINE FOR VERIFIED CODE BY CONSTRUCTION

TIAGO SILVA MEIRIM

BSc in Computer Science and Engineering

Adviser: Mário José Parreira Pereira
Assistant Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *July 10, 2025*

Abstract

The study of the Formal Verification field is very important for critical systems that involve high levels of trust. A significant advancement in recent research has been the development of certified compilers, such as CakeML, which ensure that the generated machine code preserves the behaviour of the original program. The integration of these properties in a pipeline with previously verified code can provide powerful correctness guarantees.

The main objective of this work is to explore and develop a verification pipeline that starts with programs written in OCaml with GOSPEL annotations with the end goal of producing correct-by-construction CakeML code. The first step involves translating the annotated OCaml code to WhyML using the Cameleer tool. This WhyML code should then be verified on the Why3 platform and ultimately extracted to CakeML. Additionally, the pipeline should also feature a translation scheme in the opposite direction.

Currently, the extraction mechanism in Why3 to CakeML is outdated and only supports a subset of the language. As such, we intend to revisit and expand upon it in this work, by updating the translation scheme, implement a mechanism to stop the extraction of non-verified code and to report cases where translation is not possible due to incompatibilities between features.

In this document, a thorough study was conducted concerning the state of Why3's extraction mechanism, alongside the selected tools, to identify which steps of the pipeline require modifications. Moreover, we also present the theoretical concepts concerning Operational Semantics, Deductive Verification and Verified Compilers.

Keywords: Deductive Verification, Certified Compilers, Verification Pipeline, CakeML, Cameleer, GOSPEL, OCaml, Why3

Resumo

O estudo da Verificação Formal é um campo muito importante para sistemas críticos que envolvem confiança de alto nível. Um avanço significativo na investigação recente tem sido o desenvolvimento de compiladores certificados, como o CakeML, que assegura que o código máquina gerado vai preservar o comportamento do programa original. A integração destas propriedades numa *pipeline* com código previamente verificado pode fornecer fortes garantias de correção.

O principal objetivo deste trabalho é explorar e desenvolver uma *pipeline* de Verificação onde começa com programas escritos em OCaml com anotações em GOSPEL, onde o objetivo final é produzir código CakeML correto por construção. O primeiro passo envolve a tradução de código OCaml anotado para WhyML através da ferramenta Cameleer. Este código WhyML deve ser posteriormente verificado na plataforma Why3 e em última instância extraído para CakeML. Adicionalmente, a *pipeline* deve também conter um esquema de tradução na direção oposta.

Atualmente, o mecanismo de extração do Why3 para CakeML é obsoleto e apenas suporta um subconjunto da linguagem. Como tal, tencionamos revisitar e expandí-lo neste trabalho, ao atualizar o esquema de tradução, implementar o mecanismo para parar a extração de código não verificado e reportar casos onde a tradução não é possível devido a incompatibilidades entre funcionalidades.

Neste documento, um estudo minucioso foi conduzido em relação ao estado do mecanismo de extração do Why3, em conjunto com as ferramentas escolhidas, para identificar quais os passos da *pipeline* requerem modificações. Para além disso, também apresentamos os conceitos teóricos sobre as Semânticas Operacionais, Verificação Dedutiva e Compiladores Verificados.

Palavras-chave: Verificação Dedutiva, Compiladores Certificados, *Pipeline* de Verificação, CakeML, Cameleer, GOSPEL, OCaml, Why3

Contents

List of Figures

Glossary

CakeML A functional programming language based on ML, designed with a formally verified compiler. It aims to provide a trustworthy foundation for building secure and reliable software, particularly in verified systems. *(pp. i, ii, v, 1–3, 15–24, 26–34)*

Cameleer An automated deductive verification tool for OCaml programs using GOSPEL annotations. *(pp. i, ii, v, 2, 3, 10, 11, 13, 17, 19, 20)*

CompCert A realistic and verified compiler, which enables formal reasoning and machine-checked proofs of correctness for each compilation phase. *(pp. v, 15–17)*

GOSPEL A specification language for the OCaml language, intended to be used in various purposes. The acronym stands for Generic OCaml SPEcification Language. *(pp. i, ii, 1, 2, 10–13, 17, 19–31)*

OCaml A Pragmatic functional programming language with roots in academia and growing commercial use. It supports highly complex features, such as generational garbage collection, type inference, parametric polymorphism, an efficient compiler, among many others. *(pp. i, ii, v, 1–3, 6–13, 15, 17–34)*

Why3 A platform for deductive program verification that relies on external provers. *(pp. i, ii, v, 2, 3, 5, 10, 11, 13, 17–20, 22, 24, 32)*

WhyML The programming and specification language used in the Why3 platform. It contains many features commonly present in modern functional languages, and supports built-in annotations to verification purposes. *(pp. i, ii, v, 2, 10, 11, 13, 17–20)*

1 Introduction

1.1 Motivation

Progress in Formal Verification has steadily been made over the years. The foundational ideas emerged as early as 1937 by Alan Turing [36] and by 1949 a program represented as a flowchart was formally verified [27]. Other notable papers where mathematical propositions established purely through theoretical reasoning include works by Alan Turing [37], Church [6] and Gödel [39]. These results laid important groundwork in logic, computation, and formal reasoning without relying on physical implementation or empirical methods. Significant progress was made in the 1960s with the introduction of formal systems to reason about program correctness. Floyd's work on flowcharts [12] and Hoare's in programs [14] marked a shift toward systematic verification of program behaviour using logical assertions. In recent years, two major developments have transformed the landscape of formal verification. The first is the so-called SMT [32] (Satisfiability Modulo Theories) revolution, which has enabled powerful automated verification tools capable of discharging logical obligations without human intervention [3]. The second is the emergence of certified compilers, such as CompCert [21], which provably ensures that the generated machine code faithfully preserves the semantics of the source program.

Why are verified compilers such an important target for formal verification? If a verified program is compiled by a faulty compiler, the resulting executable may not preserve its intended behaviour, invalidating any higher-level correctness obtained previously. Compilers like CompCert and CakeML [18] address this issue by providing formally verified compilation pipelines [24, 13, 22], thereby eliminating a major source of uncertainty and ensuring that correctness is preserved from source code to machine code [20].

Developments in relation to verified code have become increasingly crucial to achieve correctness and provide safety, the way we define correctness has more than one definition, it can be specified informally or written in formal language [9]. The weight of functional languages for deductive software verification has been quite low despite having good candidates for verification, like OCaml. Ever since 2018 with the introduction of GOSPEL [5], with providing formal language specification tightly integrated with OCaml, verification

has become easier with a modular specification that doesn't need many changes to OCaml code. This wasn't the first time formal logic and proof has been merged with functional programming, previous and more foundational systems like Coq [29] and WhyML [10] have paved the way. With the development of a behavioural specification language for OCaml, soon after a deductive verification tool was created for this language, that being Cameleer [30], a tool that translates a formally-specified program into corresponding code in WhyML.

With the introduction of certified compilers and state-of-the-art automated verification tools, it is possible to merge them in a pipeline to create correct high-level programs that when compiled that are not faulty. This is a highly trustworthy deductive verification system, that benefits from the features and correctness guarantees from each component of the pipeline throughout all the steps. Our motivation stems from the insurance that such a pipeline could provide to the verification of OCaml programs with GOSPEL annotations that eventually compile to correct machine code, by using the CakeML compiler.

1.2 Problem Definition

In order to create a pipeline from OCaml to CakeML and back, there is the need to understand the already existing translation tools. Firstly, Cameleer takes as input OCaml programs with GOSPEL annotations and internally these are translated onto WhyML code, which can then be verified inside Why3's ide or extracted to another language. The Why3 extraction mechanism currently allows the translation from WhyML to three other languages, with one of them being CakeML. By combining these two mechanisms we reach our goal of creating a pipeline to produce compiled code completely verified. However, we observed that the current extraction mechanism concerning CakeML is severely outdated and unreliable, after extensive tests using multiple OCaml features since it failed to compile on several instances. Cameleer also features an extraction mechanism that currently produces OCaml code from its intermediary representation, WhyML, instead of producing CakeML code. Another facet of this pipeline is to translate CakeML programs into OCaml code, and surprisingly, there is no such tool. This leads us to the following question:

How do we bridge these tools to create a pipeline that ensures correctness throughout all steps?

1.3 Goals and Expected Contributions

The first goal in this work is to have a detailed look into the features of both OCaml and CakeML languages, find the notable differences and absences, to know which features to test using the extraction mechanism. The next goal should be to change internally the source code of the Cameleer tool to support the extraction of CakeML code. With this, we should be able to identify in detail the issues with the translation, what are the necessary

changes to compile correctly, which may include introducing new features or fixing syntactic issues for the features already supported by the extraction mechanism, directly in Why3 source code. For the features that are not supported by one of the languages, providing a comprehensive error message rather than allowing the extraction to take place. To make the pipeline more robust we should confirm if the program is verified beforehand when performing an extraction. Since, there is no translation mechanism from CakeML to OCaml, it creates the need to develop a new tool that respects all the syntactic and semantic differences, meanwhile flagging with an error message any unsupported features.

1.4 Report Structure

The document is divided by 5 chapters.

In chapter 2 it is possible to find the foundational theoretical concepts for Hoare Logic and an introduction to the languages and tools used for verification and extraction mechanism.

Chapter 3 describes the relevant pieces of work already available on the context of certified compilers, while also mentioning the current verification pipeline available using Cameleer, Why3 and CakeML.

Chapter 4 displays thoroughly what the current tools do for various examples while also pointing out the places where they need improvement.

Finally, chapter 5 divides, in detail, what are the formal steps for our work and how they will be scheduled in the calendar.

2

Background

In this chapter we present the necessary background for the formal verification field and the elected tools.

2.1 Hoare logic

Hoare Logic is a formal system for reasoning about the correctness of computer programs. However, computer arithmetic often differs from the standard arithmetic familiar to mathematicians due to issues like finite precision, overflows, and machine-specific behaviors. To account for these differences, Hoare introduced a new logic [14] based on assertions and inference rules for reasoning about the partial correctness of programs. Drawing inspiration from mathematical axioms and formal proof techniques, he proposed a framework where program behavior could be specified and verified using logical formulas. This laid the foundation for systematic program verification and emphasized the need to model computational constraints, such as those arising from the limitations of machine arithmetic, within a formal system.

In Cook's seminal work [7], Hoare Logic was significantly strengthened, since Cook presented a Hoare-style axiom system tailored to a simple programming language and rigorously established both its soundness and adequacy. He concluded that, under reasonable assumptions, Hoare Logic is not only intuitively effective but also formally complete as a system for reasoning about program correctness.

2.1.1 Hoare Triples

The main construction of Hoare logic is the *Hoare triple*, where P is a pre-condition, C is a program (or a fragment) and Q is a post-condition:

$$\{P\}\ C\ \{Q\}$$

A Hoare triple expresses a partial correctness guarantee: if the pre-condition P holds before executing a program fragment C, and if C terminates, then the post-condition Q

will hold afterward. This is a partial correctness result since the termination of C is not assured by the triple. Total correctness is achieved when termination is also guaranteed.

2.1.2 Assignment Axiom

$$\frac{}{\{P_0\}\ x := f\ \{P\}} \quad (assign)$$

Where x is a variable identifier; f is an expression; P_0 is obtained from P by substituting f for all occurrences of x.

The axiom expresses that to prove a post-condition P holds after assigning the expression f to the variable x, it suffices to prove the pre-condition P_0 before the assignment, where P_0 is obtained by substituting every occurrence of x in P with the expression f.

2.1.3 Rule of Composition

The inference rule for composition states that if the post-condition of the first program segment matches the pre-condition of the second, then the entire program will produce the intended result, assuming the initial pre-condition of the first segment holds.

$$\frac{P\ \{Q_1\}\ R_1 \qquad R_1\ \{Q_2\}\ R}{P\ \{(Q_1; Q_2)\}\ R} \quad (composition)$$

2.1.4 Weakest Pre-Condition Calculus

After setting an elegant axiomatic framework for reasoning about program correctness through the use of Hoare triples $\{P\}\ C\ \{Q\}$, its practical application in large-scale or automated verification tasks presents significant challenges. Chief among these is the burden of manually identifying appropriate pre-conditions and invariants. To address this, Dijkstra [8] introduced the weakest pre-condition calculus, which reformulates program correctness into a computational problem: given a command C and a desired post-condition Q, the function $wp(C, Q)$ computes the weakest pre-condition P such that $\{P\}\ C\ \{Q\}$ holds.

This transformation from proof obligations to a calculable pre-condition function represents a critical step toward automating program verification. Unlike Hoare's original formulation, which requires deductive reasoning to derive correctness properties, weakest pre-condition semantics allow for algorithmic generation of verification conditions, thereby enabling integration with automated theorem provers and SMT solvers. The influence of weakest pre-conditions is particularly evident in modern deductive verification tools such as Why3 [4], and Dafny [19].

2.2 OCaml

OCaml is a statically typed functional programming language rooted in the ML family, originally developed to serve as the implementation language for theorem provers such as LCF. It inherits the foundational principles of typed λ-calculus, formal logic, and abstract interpretation, and extends them through practical language design aimed at enabling both expressiveness and efficiency [11].

First released in the mid-1990s, OCaml is the principal evolution of the Caml dialect of the ML family. The name OCaml, originally short for Objective Caml, reflects the addition of object-oriented features to the Caml language. While Caml stood for Categorical Abstract Machine Language, OCaml moved away from its dependence on the original abstract machine model [23]. The language is primarily developed and maintained by INRIA, which continues to guide its implementation and evolution.

OCaml distinguishes itself from many academically inspired languages through its strong emphasis on performance. Its static type system eliminates the need for runtime type checking by ensuring type correctness at compile time, thereby avoiding the performance overhead commonly associated with dynamically typed languages. This design enables OCaml to maintain high execution efficiency while preserving strong safety guarantees at runtime. Exceptions to this safety model arise only in specific low-level scenarios, such as when array bounds checking is explicitly disabled or when employing type-unsafe features like runtime serialization [11].

For the standard compiler toolchain features, OCaml has both a high-performance native-code compiler (`ocamlopt`) and a bytecode compiler (`ocamlc`). The native-code compiler produces efficient machine code via a sophisticated optimizing backend [25], while the bytecode compiler offers portability and rapid development. Both compilers are integrated with a runtime system that supports automatic memory management via a garbage collector and provides facilities for exception handling, concurrency, and system interaction.

2.2.1 Immutability by default

Immutability is promoted as a default design principle in this language. Rather than modifying existing values, new ones are created through expression evaluation. This absence of mutable shared state simplifies reasoning about program behaviour and eliminates many common sources of verification complexity, such as aliasing and unintended side effects.

OCaml

```
let x = 5
let y = x + 1 (* x is not modified, just referenced *)
```

Since values are not modified in place, program semantics are preserved under substitution, facilitating referential transparency, making symbolic execution and logical reasoning over programs more straightforward in deductive verification.

2.2.2 ADTs (Algebraic Data Types)

In OCaml, data types fall into three broad categories: atomic predefined types (e.g., `int`, `bool`), type constructors provided by the language (e.g., `list`, `array`, `option`), and user-defined types, which are declared through the general mechanism of algebraic data types, for instance:

OCaml
```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

ADTs support exhaustive pattern matching, which is particularly useful for enabling structural recursion and inductive reasoning. From a verification perspective, this ensures all possible cases are covered, making formal reasoning both precise and complete. One example of pattern matching combined with ADTs is:

OCaml
```
let rec cmp t1 t2 =
  match t1, t2 with
  | Leaf, Leaf -> true
  | Leaf, _ -> false
  | _, Leaf -> false
  | Node (l1,x1,r1), Node (l2,x2,r2) -> cmp l1 l2 && x1 = x2 && cmp r1 r2
```

The function `cmp` uses exhaustive pattern matching to compare structurally the two trees. When a tree is a leaf it means it is empty, as such, two empty trees are always equal. By contrast if one is a leaf, but the other is not, then they are different, since a node represents a non-empty tree. Finally, if both are nodes then it depends on the respective root and subtrees, in which case we have to recursively call the function `cmp` for the right and left subtrees.

2.2.3 First-Class and Higher-Order Functions

OCaml treats functions as first-class values: they can be passed as arguments, returned from other functions, and stored in data structures. Combined with lexical scoping and immutable data, this makes OCaml particularly well-suited for working with higher-order abstractions, a feature that aligns naturally with formal systems based on higher-order logic.

OCaml
```
let apply_twice f x = f (f x)
let square x = x * x
let result = apply_twice square 2  (* returns 16 *)
```

In the context of deductive verification, such functional abstractions support elegant formulations of parametric specifications and reasoning principles.

2.2.4 Side Effects in OCaml

OCaml is not a purely functional language since it allows the use of imperative constructs with side effects, like references, exceptions, built-in arrays, for and while loops. References allow the manipulation of memory, a feature not common for functional languages. A simple example about Euclidean division is displayed below:

OCaml

```
let rec eudiv_aux x y r q =
  if !r >= y then (r := !r - y; q := !q + 1; eudiv_aux x y r q)
  else ()

let euclidean_div x y =
    let r = ref x and q = ref 0 in
  eudiv_aux x y r q;
  (!q, !r)
```

The main function `euclidean_div` receives the two arguments `x` and `y`, which represent respectively the dividend and divisor. This function initializes the remainder `r` as a reference with the value of the dividend `x` and the quotient `q` as a reference of value 0. Since these variables are references, the auxiliary function may alter their in-memory value and by the end of its execution, the references `q` and `r` point to the correct values.

This example has the need for an auxiliary function, `eudiv_aux`, which receives as arguments the dividend `x`, the divisor `y`, the remainder `r`, and the quotient `q`. This function recursively increments the quotient by 1 and subtracts the remainder by the divisor, until the remainder is a non-negative value that is strictly smaller than the divisor, meaning it can not be subtracted any further, and therefore the process is finished. The values for the remainder and quotient are saved inside references and reassigned after every iteration.

OCaml also provides exception handling in a structured way to deal with runtime errors by allowing programmers to define and raise custom exceptions. In addition to user-defined exceptions, OCaml provides a range of built-in exceptions, which are extensively used by the standard library to handle common error cases. Also, it is important to mention that built-in arrays are mutable have a fixed length and allow direct access to elements using their index. We combine these two elements on the example below:

OCaml

```
let rec search_aux a c n =
  let exception Break of int in try
    if c = Array.length a then raise Not_found
    else if a.(c) = n then raise (Break c)
    else search_aux a (c+1) n
  with Break i -> i

let linear_search a n = search_aux a 0 n
```

The core function, `linear_search`, delegates the search process to the auxiliary function and initializes it with the current index as 0, as to ensure it starts at the beginning of the array. The auxiliary function `search_aux` receives as arguments an array, the current index and a value that is going to be compared to each element of the array. This function recursively traverses the array, and returns the index of the array in case the desired value was found early by raising the exception `Break`, and catching it within the `try` block. By contrast, if the value is not in the array the exception `Not_found` is raised at the end but not caught.

Overall, while OCaml encourages a functional programming style and includes strong support for immutability and higher-order functions, its imperative features make it a multi-paradigm language that combines the benefits of both functional and imperative styles.

2.2.5 Module System

The module system enables parametric modularity through modules and functors, supporting abstraction, separation of concerns, and scalable design. At the heart of this system are signatures, which serve as formal interfaces specifying the types and values a module must provide while hiding the implementation details. These signatures play a key role in formal verification by allowing reasoning about components based solely on their interfaces, without depending on how they are implemented.

This modular structure is especially useful in verification contexts because it clearly separates the interface from the implementation. This separation makes it easier to reason about and verify each component independently, improving maintainability and correctness.

We can expand on the example found in the section 2.2.2 by adding encapsulating a module around it:

OCaml

```
module Tree = struct

  type 'a tree =
    | Leaf
    | Node of 'a tree * 'a * 'a tree

  let rec cmp t1 t2 = (* ... *)
end
```

With this example we have created our own minimal `Tree` library, which includes the definition of a binary tree and the equality operation through the `cmp` function.

2.3 Why3

Why3 is the successor to the Why verification platform, offering a rich first-order language and a highly configurable toolchain for generating proof obligations in multiple formats. Its development is driven by the need to model both purely functional and imperative program behaviours and to formally verify their properties. Since verifying non-trivial programs typically requires abstracting them into pure logical models, Why3 is designed to bridge the gap between practical programming constructs and formal reasoning frameworks.

Why3 introduces WhyML, a specification and programming language that serves both as an expressive front-end and as an intermediate language for verifying programs written in other languages such as C, Java, and Ada. [10] It supports rich language features including pattern matching, recursive definitions, algebraic data types, and inductive or coinductive predicates. Moreover, it comes with a standard library of logical theories covering arithmetic, sets, maps, and more.

Why3 sets itself apart from other approaches that also provide rich specification languages like Coq and Isabelle by aiming to maximize automation. Rather than functioning as a standalone theorem prover, it serves as a front-end that generates proof obligations to be discharged by external automated provers such as Z3, Alt-Ergo, Vampire, and CVC5, as well as interactive systems like Coq and PVS.

In the context of automated program verification, Why3 simplifies the process by automatically generating verification conditions from annotated source code and delegating their resolution to a variety of powerful external theorem provers. When certain features (e.g., polymorphic types or pattern matching) are not supported by a backend prover, Why3 automatically applies transformations to encode them into a compatible form. This architecture allows developers to focus on writing correct specifications while benefiting from automation in proving correctness properties. [4]

2.4 GOSPEL and Cameleer

The evolution of deductive verification and proof automation has progressed significantly over the years. Recently, a combination of tools has been developed to apply these technologies to OCaml, a language that has not been widely explored in this context [30]. The need for a verification framework was addressed with the introduction of Cameleer, a tool for the deductive verification of programs written in OCaml, whose main objective is the automatic proof of functional code. However, Cameleer alone, using only standard OCaml, cannot perform these proofs. To enable verification, GOSPEL specifications must be added to the OCaml code.

GOSPEL terms are defined using the semantics of Separation Logic and are applied to OCaml interfaces. The acronym GOSPEL stands for "Generic OCaml Specification Language," indicating that the specification logic is not limited to a single tool but is

intended for a variety of uses, such as verification, testing, and informal documentation [5]. Unlike other behavioural specification languages such as SPARK and JML, GOSPEL supports Separation Logic, a significant extension of Hoare Logic and a powerful framework for reasoning about real-world programs [33, 28]. While GOSPEL is not the first tool to use Separation Logic, it uniquely introduces implicit permission association in data types, a feature not found in tools like VeriFast or Viper [5].

Cameleer takes as input an OCaml program annotated with GOSPEL specifications and translates it into WhyML, the intermediate language used by the Why3 platform. Once translated, the code can be processed by Why3, which leverages a range of automated theorem provers, such as Alt-Ergo, Z3, and CVC5, to discharge verification conditions. This seamless integration between Cameleer, GOSPEL, and Why3 significantly enhances the level of proof automation, allowing developers to verify functional correctness properties of OCaml programs with minimal manual intervention. This behaviour is captured by the following pipeline:

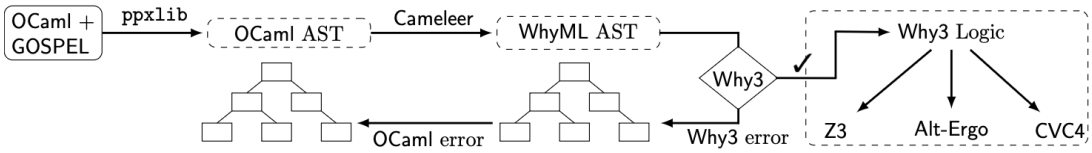

Figure 2.1: The Original Cameleer Pipeline [30]

In figure 2.1 demonstrates how the translation mechanism integrates with surrounding frameworks to produce deductively verified OCaml code annotated with GOSPEL specifications. If an error is detected in the original OCaml code during translation, the process reverts to the source, requiring corrections to ensure that the generated WhyML code is syntactically and semantically valid. Within the Why3 tool, if the generated verification goals cannot be discharged by the automated provers, it indicates that the specifications may be imprecise or incomplete and require refinement. These issues are highlighted by the tool, guiding the user to improve the annotations. Once all verification conditions are successfully proven, the resulting program is guaranteed to be free from compiler-introduced bugs or errors, ensuring a high level of formal correctness [9].

Let us take this simple example for a recursive factorial in OCaml + GOSPEL after applying the Cameleer tool:

GOSPEL + OCaml

```
let rec fact_aux n c t =
  if c <= n then fact_aux n (c+1) (t*c) else t
(*@
  r = fact_aux n c t
  requires n >= 0
  requires 0 < c <= n+1
  requires t = factorial (c-1)
  ensures r = factorial n
  variant n+1-c
```

```
*)
```

The auxiliary function `fact_aux` receives as arguments the `n` value, that represents the factorial we want to compute, the `c` value, that represents the current index of the iteration, and `t` value, that is accumulating the result from previous iterations, in order to display the solution.

The specifications include a few pre-conditions, from those, the factorial to compute is higher or equal to 0, because the factorial as default is only calculated for non-negative values, the index of the iteration is between 0 and factorial to compute, because from the code written when the index achieves the same value as the factorial to compute the function should terminate. For the post-conditions we only need to make sure the result of the auxiliary function is indeed the factorial of n, which is calculated logically with the definition below. The variant serves to prove the termination of the recursive function with an expression that decreases every iteration and is limited by 0. In this case since the counter is approaching n+1 every iteration, for the last iteration t holds the factorial of n, which was the previous value of c and terminates successfully.

In the specification of function `fact_aux` we use the logical definition of the factorial function which we present below:

GOSPEL + OCaml

```
(*@
  function rec factorial (n:int) :int =
    if n=0 then 1 else n * factorial (n-1)
*)
(*@
  requires n >= 0
  variant n
*)
```

This GOSPEL function represents the logical concept of a factorial, this can be used in other specifications to compare results. The implementation of this function is in a recursive way, since GOSPEL only supports the functional paradigm. Additionally, logical definitions are meant to be simple rather than complex but optimized, since we want to simplify the proof.

Finally, the main function `fact`:

GOSPEL + OCaml

```
let fact n = fact_aux n 1 1
(*@
  r = fact n
  requires n >= 0
  ensures r = factorial n
*)
```

Initializes the auxiliary call with appropriate base values, the first one is the counter so it needs to start at 1 because for the multiplication with the accumulator starting at 0 would not take into consideration the first iteration. For the 1 in the accumulator, the third argument, represents the neutral element for multiplication. The GOSPEL annotations

specify the expected behaviour, including pre-conditions such as non-negativity of `n` and the correctness of the returned result.

The translated code to WhyML creates 2 goals, one for the function `fact` and another for the logical function `factorial`. After splitting the goals we get for `fact` the variant and pre-condition, and for `factorial` the invariant initialization, the invariant preservation, the post-condition and the VC:

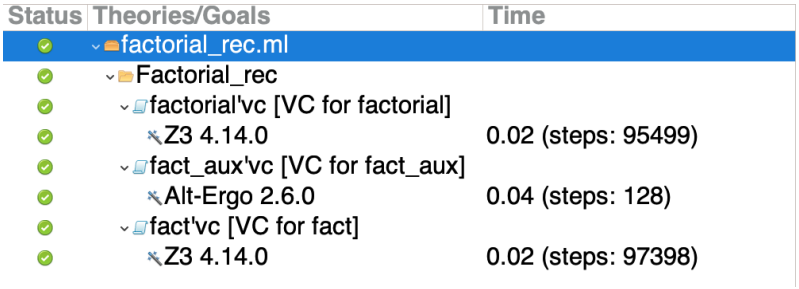

Figure 2.2: The Goals Proved in Why3

All verification goals were successfully discharged in figure 2.2 through the automated provers, resulting in the original OCaml code, annotated with GOSPEL specifications, being fully verified. This outcome demonstrates the effectiveness of the verification pipeline, where the combination of Cameleer and Why3 enables high levels of automation in proving the functional correctness of OCaml programs.

3

State Of The Art

The State-Of-The-Art outlines the most relevant and advanced work currently available in the area covered by this work. It provides an overview of existing research, tools, and methodologies, helping to frame the context in which this work is situated. By reviewing what has already been accomplished, this section highlights ongoing challenges and uncovers the gaps that this work aims to address.

3.1 Certified Compilers

A compiler is a software system that translates a program written in a source programming language into an equivalent representation in a target language, typically a lower-level language such as assembly or machine code. The goal of a compiler is not only to preserve the semantics of the original program but also to generate efficient and executable code for the target platform. [2]

A central challenge in compiler technology is: How can we trust compilers?. As discussed previously, compilers are inherently complex systems, particularly optimizing compilers, which perform intricate symbolic transformations. Despite rigorous testing practices, compilers can still introduce subtle bugs during these transformations. Such bugs are often extremely difficult to detect and diagnose [21], as they may lead to unexpected program crashes, incorrect behaviour, or silent miscomputations in the generated code, even when the original source code remains syntactically and semantically valid.

This raises serious concerns in the context of safety-critical or high-assurance software, where traditional validation through testing alone is insufficient. In such domains, testing must be complemented or in some cases replaced by formal methods, such as model checking, static analysis, and deductive program verification [21].

This is precisely where certified compilers become essential. A certified compiler is accompanied by a machine-checked formal proof that guarantees semantic preservation during the transformation from source to target code. It preserves the semantics of the source program during its transformation into target code. This approach offers strong assurances about the absence of certain classes of errors, such as compilation

bugs, compilation inaccuracies, or unsafe optimizations [22]. Unlike traditional compilers, whose correctness is typically established through empirical testing or informal reasoning, certified compilers provide mathematical guarantees of correctness, making them valuable in the development of high-assurance and safety-critical software.

3.1.1 CompCert

The development of a realistic and verified compiler began with CompCert. Here, verified denotes a compiler accompanied by a machine-checked proof that the generated code behaves exactly as prescribed by the semantics of the source program. Realistic refers to a compiler that can be effectively employed in the production of critical software systems [21].

CompCert adopts a multi-pass compilation architecture, where each pass translates an intermediate representation into a lower-level form, gradually transforming the high-level C source code into target assembly code. These intermediate languages are Clight, Cminor, RTL, LTL, and others which are formally defined within Coq. CompCert's core is implemented in Gallina, the functional programming language of Coq, which is based on the Calculus of Inductive Constructions, a powerful higher-order logic and typed λ-calculus. This implementation enables formal reasoning and machine-checked proofs of correctness for each compilation phase [26].

Although CompCert is developed in Coq, it is not executed within the proof assistant. Instead, the verified Gallina code is extracted to OCaml, where it is combined with a small portion of handwritten OCaml code to produce an efficient and executable compiler [26].

Since the compiler must generate a large subset of the C language, the code needs to be efficient enough and compact enough to fit the requirements of critical embedded systems. This implies a multi-pass compiler that features good register allocation and some basic optimizations [21]. The pipeline for the compiler is presented below:

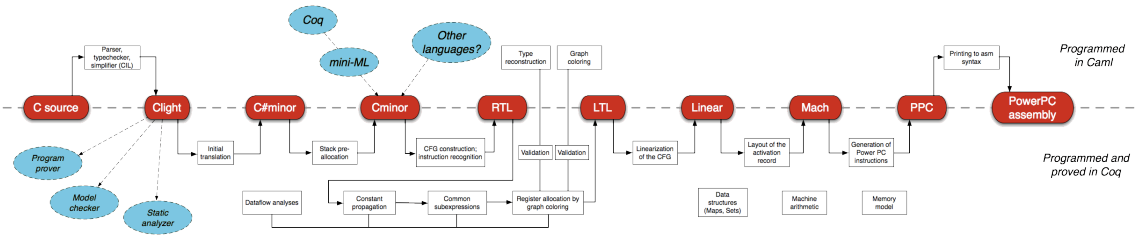

Figure 3.1: CompCert pipeline diagram [17, 1]

3.1.2 CakeML

The CakeML compiler stands out as one of the most realistic verified compilers for functional programming. Inspired by both CompCert and OCaml compilers, it supports a rich set of language features commonly found in unverified compilers for functional languages. The main goal of the CakeML compiler is to combine end-to-end formal

verification with practical performance, making it suitable for high-assurance applications without compromising efficiency [35]. CakeML is a formally verified functional programming language and compiler tool chain, specifically designed to target practical, off-the-shelf hardware while achieving a high degree of trustworthiness. It serves as a robust platform for software in domains where correctness is paramount. By focusing on functional programming and end-to-end verification, CakeML complements CompCert, which primarily target imperative languages and emphasize performance oriented optimizations. Together, these approaches address different but complementary aspects of verified software development [18].

In CakeML it is possible to find a verified type system, which is also an important achievement in the context of a certified compiler. It provides a sound and complete implementation of a system with type inference making use of the type soundness theorem and using HOL4 proof assistant. These results ensure that any CakeML program accepted by the type inferencer is guaranteed to be well-typed and thus have well-defined semantics. This contributes to CakeML's overall goal of ensuring correctness by construction throughout the pipeline [34].

CakeML can also bootstrap its compiler, meaning it can compile itself, in order to be used inside and outside the logic. This bootstrapping is performed by generating verified machine code from a formally proven semantics of the compiler, effectively closing the trusted computing base gap. This process ensures that the final executable compiler has been produced through a chain of correctness preserving transformations, all verified in the HOL4 theorem prover [35, 18].

Complementing this, CakeML adopts a multi-stage compilation pipeline, Each stage of the pipeline transforms an intermediate representation of the program into a successively lower-level form, progressing from functional language down to executable machine code. The latest verified CakeML compiler and pipeline, as of writing this document, goes through 8 intermediate languages and targets machine code for 6 architectures:

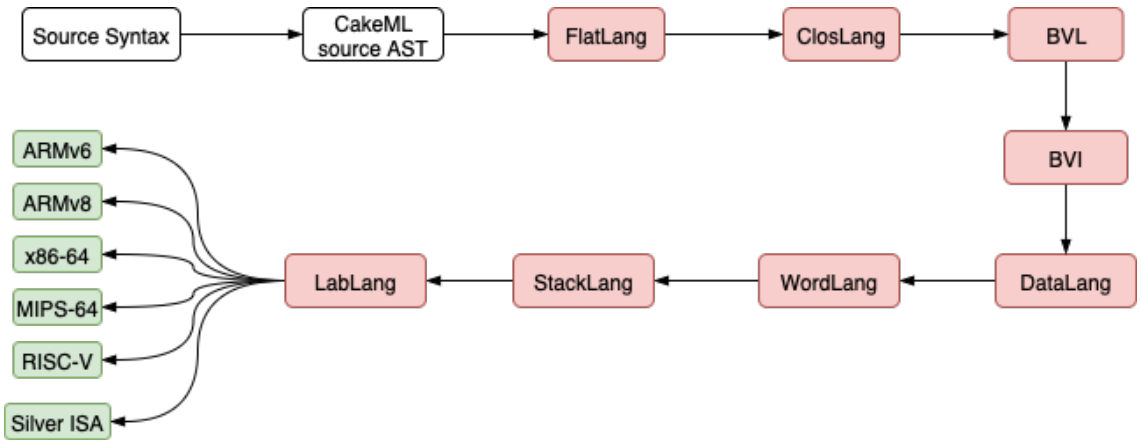

Figure 3.2: CakeML multi-stage pipeline

In the figure 3.2 the red boxes represent all the languages the compiler passes through

until it reaches the encoding for each of the different types of machine code programs, represented as green.

3.1.3 PureCake

Another tool that compiles to CakeML code with a verified compiler is PureCake, which is a HOL4-verified compiler for a lazy, functional language with Haskell-like, indentation sensitive syntax. Since, the output code is CakeML code it can be passed through CakeML trusted backend, ensuring end-to-end verification [16].

Inspired by CompCert, it targets a Standard ML-style language and compiles down to CakeML through a verified pipeline. This design not only makes it possible to compile lazy functional programs in a sound and verified way, but also showcases how flexible CakeML's verified infrastructure can be when adapting to different programming language styles. By supporting features that are rarely seen in other verified compilers, PureCake expands the range of languages that can benefit from verified compilation, showing that it's feasible to bring formal guarantees even to non-strict, high-level functional languages [16, 15].

3.2 Pipeline

In this section we reintroduce the Cameleer pipeline found in figure 2.1, but now with some slight changes to accommodate the extraction mechanism from Why3:

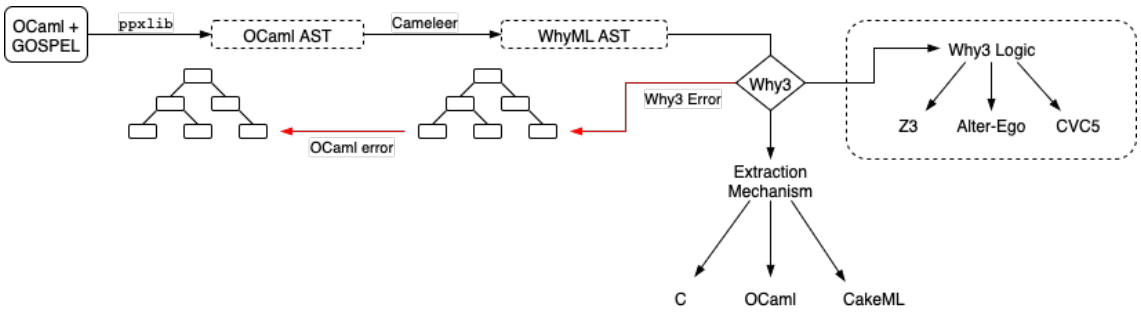

Figure 3.3: OCaml to WhyML pipeline with Cameleer

A brief explanation for the pipeline, is that Cameleer translates OCaml programs with GOSPEL annotations to WhyML language internally, on one hand if an error is caught relating to OCaml or WhyML it will be handled accordingly, on the other hand if there is no syntax error the Why3 platform is initialized and the provers can be used to verify the proof goals.

The extraction mechanism introduced in the figure 3.3 is separate from the verification process. This means that a program may be extracted to another language without requiring the correctness of the code to be proven, the only guarantee is that the program does have any OCaml or GOSPEL compilation errors. This may lead to extraction of

potentially incorrect OCaml programs, with the full responsibility of extracting after proving the code falling on the user's side.

The initial objective of the extraction mechanism was to build correct by construction libraries with the help of Why3. This lead to the creation of a mechanism that in itself does not jeopardize the behaviour of the original code when translating, this is due to its mathematical formalization [31]. One of the main advantages is the dedicated phase for the removal of ghost code and logical annotations present in the original WhyML code, in addition to a few conservative optimizations. The resulting code does not contain any logical elements which facilitates the translation to a new language, this means the only effort that needs to be done is similar to writing a printer from an intermediate representation of the language to be added [31]. Currently, four languages are allowed to be extracted from WhyML while using the Why3 platform, these are C, Java, OCaml (OCaml64) and CakeML [38, 31].

4

Preliminary Results

This chapter presents a set of manually derived examples that illustrate how OCaml + GOSPEL code can be correctly translated into semantically equivalent CakeML representations. These examples serve as a reference for the expected behaviour and structure of verified translations. The goal is to demonstrate the existing tools for translation into CakeML and identifying what the corresponding CakeML code should look like in order to compile successfully and preserve the intended behaviour.

4.1 Tool Modifications

As shown in Figure 3.3, which outlines the Cameleer pipeline, there is no explicit support for CakeML extraction. However, a small modification to the file `/bin/cli.ml` of Cameleer's source code, which allows extending the existing extraction process. By changing the `execute_extract` function to invoke `why3 extract -D cakeml %s`, the tool can target CakeML as a backend. With this adjustment, running `cameleer program.ml --extract` translates the original OCaml plus GOSPEL code into code in the CakeML language. This enables extraction from OCaml64 directly into CakeML, providing the basis for a verified compilation pipeline.

The alternative to this process is to obtain the code translated WhyML from running Cameleer with the flag `--debug` and then apply the mechanism of extraction of Why3 directly. Not only is this process more complex, but also the debug flag produces code that can not be opened by the Why3 platform because it contains proof goals directly in the code. This also implies altering the code manually or to substantially modify the behaviour of the debug flag which is not intended.

4.2 Goal Pipeline

After analysing the verification pipeline in figure 3.3 it is possible to observe that some parts have already been implemented or require only minor adjustments to meet their objectives. By modifying the extraction process in Why3 to check if the proof has been

discharged we can provide better correctness guarantees because the generated CakeML code will comply to the specification. Due to differences in syntax and features that have no direct equivalents in CakeML, we must also include some kind of error message for failures in extraction, for instance the lack of support for `while` and `for` loops.

The main goal of this work is to expand the currently available pipeline of translating code from OCaml with GOSPEL specifications into WhyML, where it can be verified using the various automated provers available in Why3. We want to achieve a more robust extraction mechanism with the ideas as previously discussed. Moreover, we ought to provide a new tool that translates compilable CakeML programs into OCaml equivalents that can be specified afterwards with GOSPEL so that one may prove their correctness in Cameleer. The figure below represents the goal pipeline:

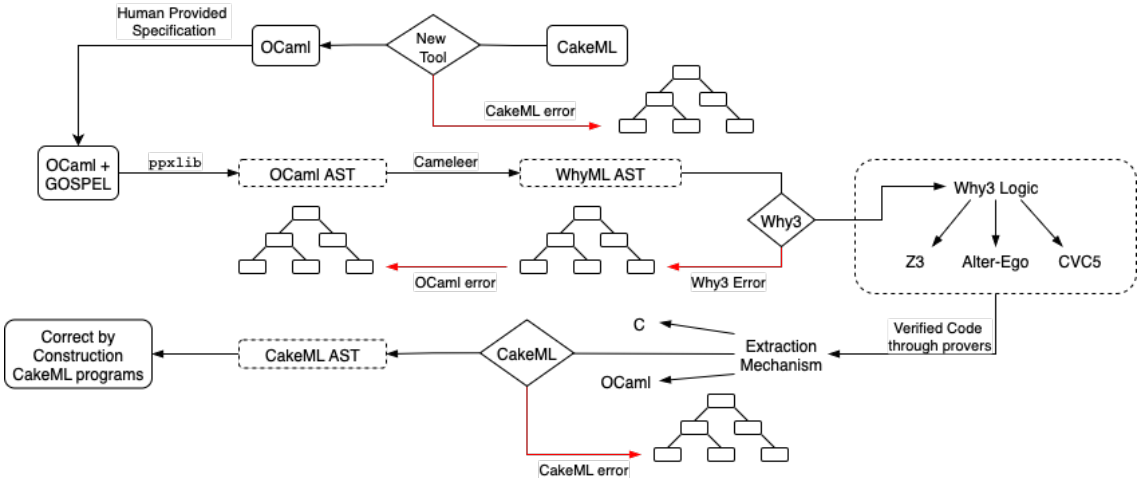

Figure 4.1: Goal pipeline

4.3 Case Studies

For all the case studies below mentioned, the OCaml + GOSPEL code was verified through Why3 automatic provers before applying the extraction command to translate into CakeML. This verification step is essential because the correctness guarantees of the final CakeML code depend entirely on the validity of the specifications and their successful discharge by the provers. Without ensuring that the WhyML intermediate code satisfies the given specifications, the generated CakeML code would no longer carry the desired formal guarantees, thereby breaking the intended end-to-end verification pipeline.

Since this step is intended only to demonstrate what already exists, the generated code may contain significant syntax errors and may not be compilable. These issues reflect limitations in the current extraction process and highlight the need for further refinement.

4.3.1 Tail Recursive Factorial

Initially an iterative version of the factorial function was also considered, however, since CakeML does not support imperative loop constructs such as `for` or `while` we had to look for alternative versions of this example. Instead, we opted to study the tail recursive variant to simulate iteration. This example highlights OCaml's support for recursion through an auxiliary function, `fact_aux`, which incrementally computes the factorial by accumulating the product of integers from 1 to n.

GOSPEL + OCaml

```
let rec fact_aux n c t =
  if c <= n then fact_aux n (c+1) (t*c) else t
(*@
  r = fact_aux n c t
  requires n >= 0
  requires 0 < c <= n+1
  requires t = factorial (c-1)
  ensures r = factorial n
  variant n+1-c
*)
```

The auxiliary function `fact_aux` receives as arguments the `n` value, that represents the factorial we want to compute, the `c` value, that represents the current index of the iteration, and `t` value, that is accumulating the result from previous iterations in order to display the solution.

The specifications include a few pre-conditions, from those, the factorial to compute is higher or equal to 0, because the factorial as default is only calculated for non-negative values, the index of the iteration is between 0 and factorial to compute, because from the code written when the index achieves the same value as the factorial to compute the function should terminate. For the post-conditions we only need to make sure the result of the auxiliary function is indeed the factorial of n, which is calculated logically with the definition below. The variant serves to prove the termination of the recursive function with an expression that decreases every iteration and is limited by 0. In this case since the counter is approaching n+1 every iteration, for the last iteration t holds the factorial of n, which was the previous value of c and terminates successfully.

GOSPEL + OCaml

```
(*@
  function rec factorial (n:int) :int =
    if n=0 then 1 else n * factorial (n-1)
*)
(*@
  requires n >= 0
  variant n
*)
```

The definition of this function in GOSPEL represents the logical concept of a factorial, this can be used for later specifications to compare results. The implementation of this function is in a recursive way, since GOSPEL only supports the functional paradigm,

additionally logical definitions are meant to be simple rather than complex but optimized, since we want to simplify the proof. The main function fact:

GOSPEL + OCaml

```
let fact n = fact_aux n 1 1
(*@
  r = fact n
  requires n >= 0
  ensures r = factorial n
*)
```

Initializes the auxiliary call with appropriate base values, the first one is the counter so it needs to start at 1 because for the multiplication with the accumulator starting at 0 would not take into consideration the first iteration. For the 1 in the accumulator, the third argument, represents the neutral element for multiplication. The GOSPEL annotations specify the expected behaviour, including preconditions such as non-negativity of `n` and the correctness of the returned result.

This annotated version allows the function to be verified in Why3 before attempting any extraction. The extracted code can be seen below:

CakeML

```
fun fact_aux n c t = let val n1 = n in
  let val c1 = c in
  let val t1 = t in
  if c1 <= n1 then (fact_aux n1 (c1 + (1)) (t1 * c1))  else (t1)

fun fact n = let val n1 = n in fact_aux n1 (1) (1)
```

The generated CakeML code failed to compile due to differences in `let` expression syntax between OCaml and CakeML. In CakeML, every `let` block must be explicitly closed with an `end` keyword, whereas in OCaml, this is not required. It should also be mentioned that the generated let-bindings generated are not necessary and only duplicate the variables in the arguments.

Initially we want to correct the let-binding termination as demonstrated in the following code:

CakeML

```
fun fact_aux n c t = let val n1 = n in
  let val c1 = c in
  let val t1 = t in
  if c1 <= n1 then (fact_aux n1 (c1 + (1)) (t1 * c1))  else (t1)
  end end end

fun fact n = let val n1 = n in fact_aux n1 (1) (1) end
```

However, our final goal is to eliminate as much unnecessary code as possible, in this case, the expendable let-bindings, while maintaining the desired behaviour:

CakeML

```
fun fact_aux n c t =
  if c <= n then (fact_aux n (c + (1)) (t * c))  else (t)

fun fact n = fact_aux n (1) (1)
```

Both versions of this example are compilable and function correctly. This first example is simple and quite clearly demonstrates that our pipeline can function if small changes are applied to the extraction mechanism.

4.3.2 Exception recursive search

One possible use of exceptions, although it may seem natural at first, can be to stop iteration early in OCaml, for instance, in the context of linear search when the desired element is found. Once again, since, for and while loops are not available in CakeML to simulate iteration we use recursion with a counter:

GOSPEL + OCaml

```
let rec search_aux a c n =
  let exception Break of int in try
    if c = Array.length a then raise Not_found
    else if a.(c) = n then raise (Break c)
    else search_aux a (c+1) n
  with Break i -> i
(*@
  r = search_aux a c n
  requires 0 <= c <= Array.length a
  requires forall k. 0 <= k < c -> a.(k) <> n
  raises Not_found -> forall k. 0 <= k < Array.length a -> a.(k) <> n
  ensures 0 <= r < Array.length a
  ensures a.(r) = n
  variant Array.length a - c
*)
```

The auxiliary function `search_aux` receives as arguments an array, the current index and a value that is going to be compared to each element of the array. The program returns the index of the array in case it was found early by raising the exception `Break`. By contrast, if the value is not in the array the exception `Not_found` is raised at the end.

Inside the GOSPEL specifications, we must guarantee that index `c` is between values of 0 and the length of the array `a`, which in turn gives us the assurance that the array is never accessed for values out of its range expect for the case where `c` is equal to the length in which case the function terminates without accessing the array. To further strengthen the precondition above there is the need to state that all the values previously seen are not the desired value, otherwise the function should have terminated earlier.

Since the exception `Not_Found` was not handled this means that the function does not return a value in case it is raised, therefore only when a value is found it terminates successfully. In that case we need to express two ensure clauses, one that states the resulting index is a value between 0 and the length of the array a minus one and that the element of the array in the index returned is effectively the value searching for. As the index is increasing in each iteration, at some point it becomes equal to the length of the array, thereby proving termination and the variant clause.

The core function, `linear_search`, delegates the search process to the auxiliary function that recursively traverses the array:

GOSPEL + OCaml

```
let linear_search a n = search_aux a 0 n
(*@
  r = linear_search a n
  raises Not_found -> forall k. 0 <= k < Array.length a -> a.(k) <> n
  ensures 0 <= r < Array.length a
  ensures a.(r) = n
*)
```

Following the same train of thought, we must express the two possible outcomes. The first is when the `Not_found` exception is raised and the element was not found within the boundaries of the array. The second case is when the result is in fact an integer inside the boundaries and the value inside the array in the index provided is the same value being searched.

This makes the function verifiable by Why3 before attempting extraction to CakeML, ensuring that key correctness properties are formally validated. The extracted code is:

CakeML

```
fun search_aux a c n = let val a1 = a in
  let val c1 = c in
  let val n1 = n in
  let exception Break of (int) in
  ((if c1 = (a1.length) then (raise Not_found)
    else (if (get a1 c1) = n1 then (raise (Break n1))
          else (search_aux a1 (c1 + (1)) n1)))
  handle   Break i => i)

fun linear_search a n =
  let val a1 = a in let val n1 = n in search_aux a1 (0) n1
```

The generated CakeML code fails to compile due to several incompatibilities between OCaml and CakeML that must be addressed to ensure a successful and semantically correct extraction. One of the primary issues concerns exception handling, as CakeML requires all exceptions to be declared globally, outside the scope of functions, whereas the extracted code attempts to define the exception Break locally within a function using let-bound exception, a construct that is not valid in CakeML. This syntactic difference results in a compilation error and reflects a broader incompatibility in exception scoping between the two languages.

Additionally, array operations in CakeML differ substantially from those in OCaml. While OCaml allows for concise access through expressions like `a.(i)` while CakeML demands the usage of a function from the array library `Array.sub a i`. The generated code fails in this aspect by using invalid expressions such as `a1.length` and `get a1 c1`, neither are supported by the CakeML standard library nor array library. This divergence in the array library further contributes to the failure of the generated code to compile.

Moreover, the problems with let-bindings discussed earlier also contribute to another critical issue. These problems collectively illustrate the importance of a properly configured

and semantically aware translation mechanism. The corrected code is presented below:

CakeML

```
exception Not_found
exception Break int

fun search_aux a c n = let val a1 = a in
  let val c1 = c in
  let val n1 = n in
  ((if c1 = (Array.length a1) then (raise Not_found)
    else (if (Array.sub  a1 c1) = n1 then (raise (Break n1))
          else (search_aux a1 (c1 + (1)) n1)))
  handle Break i => i)
  end end end

fun linear_search a n =
  let val a1 = a in let val n1 = n in search_aux a1 (0) n1
  end end
```

With this example we wanted to showcase how the extraction mechanism translates array definitions, access and other operations as well as handling exceptions.

4.3.3 Count Even For A List

Since operations for arrays have already been explored in the previous case study, we also want to explore if the same happens with operations for lists. We also utilize the mod operator from the standard library, when trying to count every even number inside a given list:

GOSPEL + OCaml

```
let count_even l =
  let l1 = List.filter (fun x -> x > 0 && x mod 2 = 0) l in
  let l2 = List.rev l1 in
  (List.length l2, l2)
(*@ (s, r) = count_even l
    ensures s = List.length r
    ensures forall x: int. List.mem x r -> x > 0 && mod x 2 = 0
    ensures forall x: int. List.mem x r -> List.mem x l
    ensures forall x: int. not List.mem x l -> not List.mem x r *)
```

The function `count_even` receives as the sole argument the list to be searched. The following code for the `l1` is to apply a filter for the original list `l`, where inside the filter we only want values that are positive and even. Theoretically, the operation is already done, but for research purposes we added more operations to see how the translation would undergo for different operations, those where the inversion of the list and the length of a list.

The specification passes through ensuring some crucial information. As the function is organized, it returns a tuple with both the corresponding length of the list created and the list created, so, we start by comparing the first element of the tuple, which should be

the length of the reversed and filtered list, and therefore equal to the length of the second argument, the same list. Then there is a need to guarantee all the values inside the created list are effectively positive and even. Additionally, every element belonging inside the list created is also inside the original list and every value that is not inside the original list is not inside the created list. For simplicity, we do not guarantee that the number of occurrences for each number that belong to the resulting list is also the same as the original list. This condition would guarantee of this function however its proof would be complex to display here.

The extracted code can be seen below:

CakeML

```
fun count_even l =
  let val l1 = l in
  let val l11 = filter (fn x => (x > (0)) andalso ((mod x (2)) = (0))) l1 in
  let val l2 = rev l11 in (List.length l2, l2)
```

The list operations are not correctly declared, it seems only the length operation is written with the correct syntax. The mod operator is used as if it is a function from the standard library, however in CakeML it is part of the `Int` library. The duplicate let-bindings were also generated for every argument.

Correcting the code, becomes the block below:

CakeML

```
fun count_even l =
  let val l1 = List.filter (fn x => (x > (0)) andalso ((Int.mod x (2)) = (0))) l in
  let val l2 = List.rev l1 in (List.length l2, l2) end end
```

Using the correct syntax for calling every list operation and integer operation while deleting all the unnecessary let-bindings makes the code compilable and correct.

4.3.4 Depth-search tree

User defined data types, such as trees, are very important for real world examples in functional programming, since traversing them can easily be achieved recursively.

GOSPEL + OCaml

```
type 'a tree =
    | Leaf
    | Node of 'a tree * 'a * 'a tree
```

A tree can either be a `Leaf`, which represents the empty tree, or a `Node`, which is a tuple with three elements, the subtree to the left, the value for the node and the subtree to the right, respectively. The recursive depth-first search algorithm for trees can be implemented as:

GOSPEL + OCaml

```
let rec depth_search t n =
  match t with
  | Leaf -> false
  | Node (l,x,r) -> x = n || depth_search l n || depth_search r n
(*@
  r = depth_search t n
  variant t
```

```
  ensures List.mem n (to_list t) <-> r
*)
```

The following recursive function `depth_search` shows how a tree is traversed through its depth firstly when trying to find a node that has the same value as the value given in the arguments.

Since the algorithm searches the subtree to the left and subtree to the right it is known that each of those trees is smaller than their parent. In the worst case, where the element is not found in the tree, we know that recursion terminates with the leaves, so the variant clause is the tree that was searched. One way to guarantee the result is correct is to transform the tree into a list and checking if the desired element is found in the resulting list with the `List.mem` operation.

To transform the tree into a list we formed an auxiliary logical function:

GOSPEL + OCaml

```
(*@ function rec to_list (t: 'a tree) : 'a list =
  match t with
  | Leaf -> []
  | Node l x r -> x :: to_list l @ to_list r
*)
(*@
  variant t
*)
```

The recursive function `to_list` transforms a tree firstly from the own element of the root, following the left side and ending with the right side. Since, for this example, we are only concerned with the inclusion of an element in the list, the order in which it is constructed doesn't have effect on the outcome of the solution, our decision was arbitrary.

Similarly to the function `depth_search`, the termination can be proven by the tree itself.

The generated CakeML code was:

CakeML

```
'a datatype tree = Leaf | Node of 'a tree * 'a * 'a tree

fun depth_search t n = let val t1 = t in
  let val n1 = n in
  (case t1 of
    Leaf => false
  | Node l x r =>
    (x = n1) orelse ((depth_search l n1) orelse (depth_search r n1)))
```

The definition of the tree data type is incorrect for three reasons. Number one the ordering of the syntax for the left side declaration of a data type starts with the token `datatype` and only after the generic type `'a`. The second reason is the unnecessary token `of` that is not used when declaring a tuple in CakeML syntax. Thirdly, the tokens `*` are also unnecessary when declaring tuples. Finally, the let-bindings displayed the same errors as previous examples.

The corrected code below:

CakeML

```
datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)
```

```
fun depth_search t n = let val t1 = t in
  let val n1 = n in
  (case t1 of
    Leaf => False
  | Node l x r =>
    (x = n1) orelse ((depth_search l n1) orelse (depth_search r n1)))
    end end
```

Correcting the order and removing the unnecessary tokens in data type definition, achieves a more robust extraction mechanism that produces compilable CakeML code.

4.3.5 Tree Comparison

With this example we only want to understand if the module system is working as intended in CakeML. The module system in CakeML is very limited since it does not support signatures and functors. We present a simple tree module in OCaml that contains the comparison between trees.

GOSPEL + OCaml

```
module Tree = struct

  type 'a tree =
    | Leaf
    | Node of 'a tree * 'a * 'a tree

  let rec cmp t1 t2 =
    match t1, t2 with
    | Leaf, Leaf -> true
    | Leaf, _ -> false
    | _, Leaf -> false
    | Node (l1,x1,r1), Node (l2,x2,r2) -> cmp l1 l2 && x1 = x2 && cmp r1 r2

  (*@
  r = cmp t1 t2
  variant t1
  ensures r <-> t1 = t2
  *)
end
```

Logically, the result should be equal to comparing `t1` and `t2` and the variant clause could be either one of the trees since it only matters when the result is correct therefore the trees have the same structure, and in each recursive call we are considering fewer elements until, eventually, there are no more elements.

Generated code to CakeML was:

CakeML

```
'a datatype tree = Leaf | Node of 'a tree * 'a * 'a tree

fun cmp t1 t2 = let val t11 = t1 in
  let val t21 = t2 in
  (case (t11, t21) of
```

```
    (Leaf, Leaf) => true
  | (Leaf, _) => false
  | (_, Leaf) => false
  | (Node l1 x1 r1, Node l2 x2 r2) =>
    (cmp l1 l2) andalso ((x1 = x2) andalso (cmp r1 r2)))
```

Once more, the data types and let-bindings expressions failed to extract correctly, but more notably the module is not present. The corrected version is demonstrated below:

CakeML

```
structure Tree = struct

    datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)

    fun cmp t1 t2 =
    (case (t1, t2) of
        (Leaf, Leaf) => True
    | (Leaf, _) => False
    | (_, Leaf) => False
    | (Node l1 x1 r1, Node l2 x2 r2) =>
        (cmp l1 l2) andalso ((x1 = x2) andalso (cmp r1 r2)))

end
```

For the same problems, the same solutions were implemented, While modules are not explicitly shown in CakeML's documentation, by having a look at SML's module syntax, it was possible to reach the `structure Tree = struct ... end` construct, which was not presented in the translated code.

4.3.6 References

To demonstrate support for references, a program was implemented to perform Euclidean division, also known as division with remainder. This operation involves dividing one non-negative integer by another positive integer, resulting in two values: a positive integer quotient and a non-negative remainder that is strictly less than the absolute value of the divisor. This program is implemented below:

GOSPEL + OCaml

```
let rec eudiv_aux (x: int) (y: int) (r: int ref) (q: int ref) =
  if !r >= y then (r := !r - y; q := !q + 1; eudiv_aux x y r q)
  else ()
(*@ res = eudiv_aux x y r q
    requires x >= 0
    requires y > 0
    requires 0 <= !r <= x
    requires x = y * !q + !r
    variant !r - y
    ensures x = y * !q + !r
    ensures 0 <= !r <= x
    ensures 0 <= !r < y *)
```

This example has the need for an auxiliary function in `eudiv_aux` which receives as arguments the values the dividend `x`, the divisor `y`, the remainder `r`, and the quotient `q`. What the recursive auxiliary function is doing for every iteration is to increment the quotient by 1 and subtract the remainder by the divisor, until the remainder is a non-negative value that is strictly smaller than the divisor. The values for the remainder and quotient are saved inside references and reassigned after every operation or iteration.

To specify this function, we used GOSPEL annotations to define pre-conditions and post-conditions that ensure the correctness of the Euclidean division. The pre-conditions enforce that the dividend must be a non-negative integer, the divisor must be strictly positive, and the remainder (held in a reference) must initially lie between 0 and the dividend. Additionally, we specify that the dividend is equal to the product of the divisor and quotient plus the remainder at each recursive call. For the post-conditions, we ensure that this equation is preserved after execution. Furthermore, the remainder must always remain non-negative and strictly less than the divisor, which aligns with the definition of Euclidean division. In every iteration, we subtract `y` from the remainder `r` so a natural way to guarantee termination is to use this exact expression as a variant condition.

GOSPEL + OCaml

```
let euclidean_div x y =
    let r = ref x and q = ref 0 in
  eudiv_aux x y r q;
  (!q, !r)
(*@ (q, r) = euclidean_div x y
    requires x >= 0
    requires y > 0
    ensures x = y * q + r
    ensures 0 <= r < y *)
```

The main function `euclidean_div` receives the two arguments `x` and `y`, which represent respectively the dividend and divisor. This function initializes the remainder `r` as a reference with the value of the dividend `x` and the quotient `q` as a reference of value 0. After acquiring the necessary variables the auxiliary function is called.

The GOSPEL specifications require the dividend to be a non-negative integer and the divisor to be strictly positive. The function ensures that the dividend always equals the product of the divisor and the quotient plus the remainder. Additionally, it guarantees that the remainder remains non-negative and strictly less than the divisor, as defined by the Euclidean division properties.

The translated code to CakeML is represented below:

CakeML

```
fun eudiv_aux x y r q = let val x1 = x in
  let val y1 = y in
  let val r1 = r in
  let val q1 = q in
  if (!r1) >= y1
  then ((r1 := ((!r1) - y1); q1 := ((!q1) + (1)); eudiv_aux x1 y1 r1 q1))

fun euclidean_div x y =
```

```
  let val x1 = x in
  let val y1 = y in
  let val r = ref x1 in
  let val q = ref (0) in (eudiv_aux x1 y1 r q; (!q, !r))
```

The operations concerning references, creation, assignment and access, all seem to be well translated syntactically, the problem with let-bindings is recurring, however a new issue arises as the `else` branch with a single unit value was not generated for the function `eudiv_aux`.

The following code for CakeML is the corrected version:

CakeML

```
fun eudiv_aux x y r q =
  if (!r) >= y
  then ((r := ((!r) - y); q := ((!q) + (1)); eudiv_aux x y r q))
  else ()

fun euclidean_div x y =
  let val r = Ref x in
  let val q = Ref (0) in (eudiv_aux x y r q; (!q, !r))
end end
```

Removing the unnecessary let-bindings and correctly handling of the `else` branch, which was absent, the functions became both syntactically correct and successfully compilable.

4.4 Analyses For The Case Studies

During our studies we found various issues, such as exception scoping, array operation semantics, and let-binding syntax, the extracted code remains invalid and cannot serve as a verified executable target. Therefore, improving the extraction pipeline from OCaml and GOSPEL to CakeML necessitates careful handling of these language specific constructs to ensure that the resulting code is not only syntactically correct but also preserves the behaviour of the original verified source.

5

Work Plan

In this chapter we enumerate all the steps for the next few months about the development and objectives to be done for our work. Every task is put onto a flexible time frame, because some tasks may end sooner or later than we expect, leading us to adjust the schedule when needed.

5.1 From OCaml to CakeML

The first part for this work is focusing on improving the extraction mechanism, while fixing the issues aforementioned.

- **Correct Boolean Literals Syntax -** In CakeML, boolean literals start with uppercase contrary to OCaml. In this task we ought to revise the Why3's extraction scheme, which currently define these literals with all lowercase letters. So, in order to match boolean literals with CakeML syntax, we must capitalize the initial letters (e.g., `true` → `True`).

- **Correct Reference Declaration Syntax -** Similar to boolean literals, reference declarations also start with an uppercase letter, which contrasts with OCaml. The same capitalization solution will be applied to the reference token in order to follow CakeML conventions (e.g., `ref` → `Ref`).

- **Correct Let-Binding Syntax -** Regarding the let-binding structure in CakeML, this always terminates with the `end` keyword. During translation sometimes there are some missing `end` tokens. This may be due to OCaml not having an explicit termination token for let-bindings. Additionally, the parameters of the functions are duplicated with let-bindings, this is redundant, and it is our objective to simplify the resulting translations.

- **Correct Data Structures Operations Syntax -** Two of the most commonly used libraries for data structures, List and Array, have displayed errors when translating their operations into CakeML. Some of the errors include the omitted prefix for the

library name and different operation names. Even if the operations are quite similar from OCaml and CakeML, the translation doesn't generate the same operation calling, therefore, to have a fully correct and compilable code in CakeML arises the necessity to replace the data structures operations with the corresponding ones.

- **Correct Data Type Syntax -** When defining the data types, CakeML's syntax differs from OCaml's when it comes to tuple declarations, in particular, the `of` keyword and the separator `*` found in OCaml are not used in CakeML. Another translation error consists in misplacement of the generic type, which is situated before the `datatype` keyword, rather than after. The solution comes through fixing data type definitions to respect CakeML ordering and syntax.

- **Correct Exception Handling -** In OCaml we can declare exceptions within functions through let-bindings, contrary to CakeML. All exceptions need to be declared at the global level. Furthermore, OCaml supports a few predefined exceptions found in the standard library which have no direct translation in CakeML, so they also need to be declared. For this task it is essential to refactor the exception declarations and usage to comply with CakeML, ensuring all exceptions are defined properly.

5.2 Translation from CakeML to OCaml

The next big task is to create a new tool that can do the inverse pipeline of the previous work. This new tool will respect the syntactic and semantic differences while translating, and the available features of CakeML.

- **Arithmetic and Boolean operations -** The first step in any basic translation tool is to define the integer and boolean data types as well as their respective operations. This includes defining the literals correctly, as previously mentioned the boolean tokens are not equally written, and operations such as addition, subtraction, multiplication, equality and inequality. Whenever necessary, casing and syntax will be adjusted.

- **Let-Bindings -** This construct is essential to functional languages. The correct syntax for this construct in CakeML includes explicit termination with the `end` keyword, meanwhile in OCaml there is no explicit termination. This construct requires special attention when multiple let-bindings are nested due to the different environments.

- **References -** To support references it is important to define three operations, these being creation, assignment and access. The correct translation from CakeML to OCaml must take into consideration the main syntax difference which is casing: `ref` in OCaml and `Ref` in CakeML.

- **Data Structure Operations -** Most real programs make use of built-in complex data structures such as Arrays and Lists. This urges for the translation of their basic

operations already present in the standard library, besides the additional operations featured in the respective dedicated library for that data structure.

- **Data Types -** For many programs the built-in data structures do not suffice, so, user defined data types are a must for programming languages. Such that, it is natural to provide a translation scheme for the definition of data types in CakeML into idiomatic OCaml type declarations, preserving constructors and structure.

- **Exception Handling -** Robust programs are built with the expectation to fail occasionally, so there should be mechanisms to handle and recover from these situations. One of these mechanisms are exceptions, which offer clean and concise flagging for these non-natural cases. When transforming exception declarations and usage from CakeML back into OCaml style with inline definitions and raises, we must also take into consideration that CakeML only allows global declaration of exceptions.

5.3 Dissertation

- **Writing -** In the last month, we will finalize the dissertation document by detailing the modifications made to the codebase, describing the implementation steps, and analysing the results and limitations of the pipeline and the translation tool.

5.4 Gantt Chart

Below we present a Gantt Chart with the previously mentioned tasks spread into the next seven months.

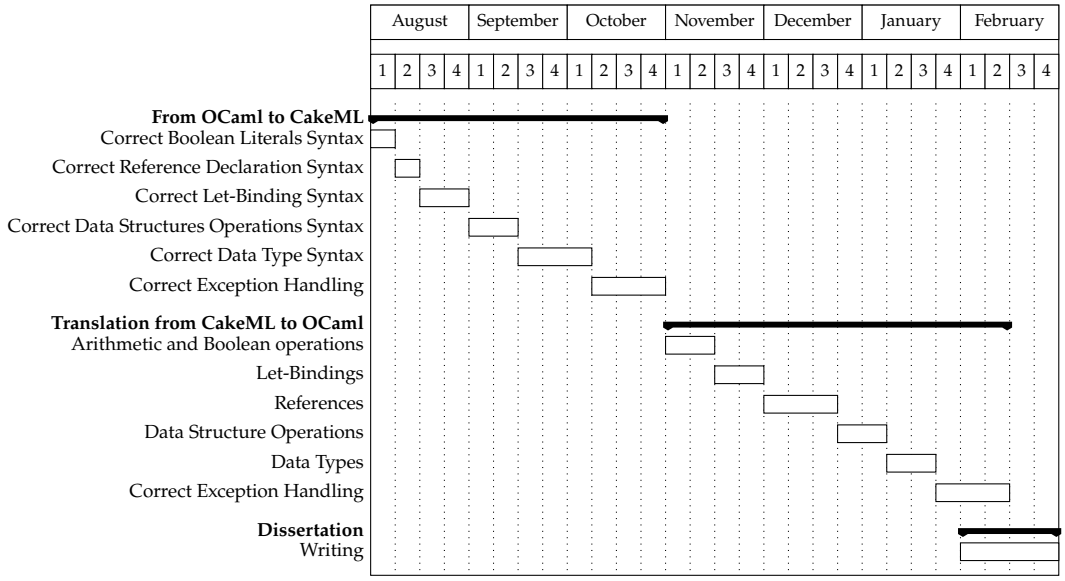

Figure 5.1: Tentative Schedule

Bibliography

[1] *"The structure of CompCert"*. Accessed: 2025-07-10. URL: https://www.absint.com/compcert/structure.htm (cit. on p. 15).

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: https://www.worldcat.org/oclc/12285707 (cit. on p. 14).

[3] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by E. M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11 (cit. on p. 1).

[4] F. Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. https://hal.inria.fr/hal-00790310. Wrocław, Poland, 2011-08, pp. 53–64 (cit. on pp. 5, 10).

[5] A. Charguéraud et al. "GOSPEL - Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by M. H. ter Beek, A. McIver, and J. N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. DOI: 10.1007/978-3-030-30942-8_29. URL: https://doi.org/10.1007/978-3-030-30942-8%5C_29 (cit. on pp. 1, 11).

[6] A. Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936-04), pp. 345–363. ISSN: 00029327. DOI: 10.2307/2371045. URL: http://dx.doi.org/10.2307/2371045 (cit. on p. 1).

[7] S. A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005. eprint: https://doi.org/10.1137/0207005. URL: https://doi.org/10.1137/0207005 (cit. on p. 4).

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://www.worldcat.org/oclc/01958445 (cit. on p. 5).

[9] J. Filliâtre. "Deductive software verification". In: *Int. J. Softw. Tools Technol. Transf.* 13.5 (2011), pp. 397–403. DOI: 10.1007/S10009-011-0211-0. URL: https://doi.org/10.1007/s10009-011-0211-0 (cit. on pp. 1, 11).

[10] J. Filliâtre and A. Paskevich. "Why3 - Where Programs Meet Provers". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by M. Felleisen and P. Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_8 (cit. on pp. 2, 10).

[11] J.-C. Filliâtre, M. Pereira, and S. M. de Sousa. "A Toolchain to Produce Correct-by-Construction OCaml Programs". In: *HAL Working Paper* (2018). Preprint, May 2018. URL: https://hal.science/hal-01783851v1/file/main.pdf (cit. on p. 6).

[12] R. W. Floyd. "Assigning Meanings to Programs". In: *Program Verification - Fundamental Issues in Computer Science*. Ed. by T. R. Colburn, J. H. Fetzer, and T. L. Rankin. Vol. 14. Studies in Cognitive Systems. Springer Netherlands, 1993, pp. 65–81. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7%5C_4 (cit. on p. 1).

[13] J. Gross et al. "Accelerating Verified-Compiler Development with a Verified Rewriting Engine". In: *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*. Ed. by J. Andronick and L. de Moura. Vol. 237. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 17:1–17:18. DOI: 10.4230/LIPICS.ITP.2022.17. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.17 (cit. on p. 1).

[14] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on pp. 1, 4).

[15] H. Kanabar, K. Korban, and M. O. Myreen. "Verified Inlining and Specialisation for PureCake". In: *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*. Ed. by S. Weirich. Vol. 14577. Lecture Notes in Computer Science. Springer, 2024, pp. 275–301. DOI: 10.1007/978-3-031-57267-8_11. URL: https://doi.org/10.1007/978-3-031-57267-8%5C_11 (cit. on p. 17).

[16] H. Kanabar et al. "PureCake: A Verified Compiler for a Lazy Functional Language". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 952–976. DOI: 10.1145/3591259. URL: https://doi.org/10.1145/3591259 (cit. on p. 17).

[17] D. Kästner et al. "CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler". In: *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE. Toulouse, France, 2018-01, pp. 1–9. URL: https://inria.hal.science/hal-01643290 (cit. on p. 15).

[18] R. Kumar et al. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages (POPL)*. ACM Press, 2014-01, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/popl14.pdf (cit. on pp. 1, 16).

[19] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by E. M. Clarke and A. Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://doi.org/10.1007/978-3-642-17511-4%5C_20 (cit. on p. 5).

[20] X. Leroy. *"Formally verifying a compiler: what does it mean, exactly?"* Accessed: 2025-07-03. 2016. URL: https://xavierleroy.org/talks/ICALP2016.pdf (cit. on p. 1).

[21] X. Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/S10817-009-9155-4. URL: https://doi.org/10.1007/s10817-009-9155-4 (cit. on pp. 1, 14, 15).

[22] X. Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814 (cit. on pp. 1, 15).

[23] X. Leroy. *The ZINC experiment : an economical implementation of the ML language*. Tech. rep. RT-0117. INRIA, 1990-02, p. 100. URL: https://inria.hal.science/inria-00070049 (cit. on p. 6).

[24] A. Lööw et al. "Verified compilation on a verified processor". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM, 2019, pp. 1041–1053. DOI: 10.1145/3314221.3314622. URL: https://doi.org/10.1145/3314221.3314622 (cit. on p. 1).

[25] B. Meurer. "OCamlJIT 2.0 - Faster Objective Caml". In: *CoRR* abs/1011.1783 (2010). arXiv: 1011.1783. URL: http://arxiv.org/abs/1011.1783 (cit. on p. 6).

[26] D. Monniaux and S. Boulmé. "The Trusted Computing Base of the CompCert Verified Compiler". In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by I. Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 204–233. DOI: 10.1007/978-3-030-99336-8_8. URL: https://doi.org/10.1007/978-3-030-99336-8%5C_8 (cit. on p. 15).

[27] F. L. Morris and C. B. Jones. "An Early Program Proof by Alan Turing". In: *IEEE Ann. Hist. Comput.* 6.2 (1984-04), pp. 139–143. ISSN: 1058-6180. DOI: 10.1109/MAHC.1984.10017. URL: https://doi.org/10.1109/MAHC.1984.10017 (cit. on p. 1).

[28] P. W. O'Hearn, J. C. Reynolds, and H. Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by L. Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1. URL: https://doi.org/10.1007/3-540-44802-0%5C_1 (cit. on p. 11).

[29] C. Paulin-Mohring. "Extracting omega's programs from proofs in the calculus of constructions". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 89–104. ISBN: 0897912942. DOI: 10.1145/75277.75285. URL: https://doi.org/10.1145/75277.75285 (cit. on p. 2).

[30] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_31 (cit. on pp. 2, 10, 11).

[31] M. J. P. Pereira. "Tools and Techniques for the Verification of Modular Stateful Code. (Outils et techniques pour la vérification de programmes impératives modulaires)". PhD thesis. University of Paris-Saclay, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343 (cit. on p. 18).

[32] A. Reynolds and C. Tinelli. "SyGuS Techniques in the Core of an SMT Solver". In: *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. Ed. by D. Fisman and S. Jacobs. Vol. 260. EPTCS. 2017, pp. 81–96. DOI: 10.4204/EPTCS.260.8. URL: https://doi.org/10.4204/EPTCS.260.8 (cit. on p. 1).

[33] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. URL: https://doi.org/10.1109/LICS.2002.1029817 (cit. on p. 11).

[34] Y. K. Tan, S. Owens, and R. Kumar. "A verified type system for CakeML". In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, Koblenz, Germany, September 14-16, 2015*. Ed. by R. Lämmel. ACM, 2015, 7:1–7:12. DOI: 10.1145/2897336.2897344. URL: https://doi.org/10.1145/2897336.2897344 (cit. on p. 16).

[35] Y. K. Tan et al. "The verified CakeML compiler backend". In: *J. Funct. Program.* 29 (2019), e2. DOI: 10.1017/S0956796818000229. URL: https://doi.org/10.1017/S0956796818000229 (cit. on p. 16).

[36] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proc. London Math. Soc.* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230. URL: https://doi.org/10.1112/plms/s2-42.1.230 (cit. on p. 1).

[37] A. M. Turing. "Systems of Logic Based on Ordinals". PhD thesis. Princeton University, NJ, USA, 1938. DOI: 10.1112/PLMS/S2-45.1.161. URL: https://doi.org/10.1112/plms/s2-45.1.161 (cit. on p. 1).

[38] *Why Extraction Command*. Accessed: 2025-07-10. URL: www.why3.org/doc/manpages.html#why3:tool-extract (cit. on p. 18).

[39] R. Zach. "Chapter 71 - Kurt Gödel, paper on the incompleteness theorems (1931)". In: *Landmark Writings in Western Mathematics 1640-1940*. Ed. by I. Grattan-Guinness et al. Amsterdam: Elsevier Science, 2005, pp. 917–925. ISBN: 978-0-444-50871-3. DOI: https://doi.org/10.1016/B978-044450871-3/50152-2. URL: https://www.sciencedirect.com/science/article/pii/B9780444508713501522 (cit. on p. 1).