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 7, 2025*

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 7, 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Currently, the extraction mechanism in Why3 to CakeML is outdated and only supports a subset of the language.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

A verificação formal de programas é uma área muito importante, especialmente em sistemas críticos que exigem alto grau de confiança. Ferramentas como Why3 permitem a especificação e verificação automática de programas através da geração de condições de verificação, que são posteriormente delegadas a provadores externos. Por outro lado, CakeML oferece uma cadeia de compilação completamente verificada, baseada numa semântica operacional formal de Standard ML, permitindo uma verificação de ponta a ponta, desde o código fonte com anotações formais até código executável com garantias formais.

O principal objetivo desta dissertação é explorar uma pipeline de verificação que parte de programas escritos em OCaml, anotados com especificações em GOSPEL, passando pela tradução automática para WhyML, onde a verificação é efetuada, culminando na geração de código equivalente em CakeML. Pretende-se garantir que o código extraído preserva as propriedades verificadas, promovendo uma verificação formal contínua ao longo de todo o processo. Adicionalmente, a pipeline vai suportar tradução de códigoem CakeML para código em OCaml.

O trabalho a realizar envolve a análise e adaptação da ferramneta de extração já existente no Cameleer, de forma a permitir a conversão segura de OCaml para CakeML, respeitando as diferenças sintáticas e semânticas entre as linguagens. Serão também implementados mecanismos de deteção e sinalização de casos em que a tradução não é possível devido a incompatibilidades entre os modelos de execução.

Na fase de preparação, foi realizado um estudo aprofundado sobre semânticas operacionais formais, na lógica de Hoare, na utilização de pré-condições fracas e compiladores verificados. Suplementarmente, o estudo do funcionamento interno das ferramentas Why3, Cameleer e CakeML serviu para identificar quais as etapas da pipeline que requerem alterações ou integração.

Palavras-chave: Primeira palavra-chave, Outra palavra-chave, Mais uma palavra-chave, A última palavra-chave

Contents

List of Figures **v**

Glossary **vi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 2

1.3 Goals and Expected Contribution 2

1.4 Report Structure 2

2 Background **3**

2.1 Hoare logic 3

2.1.1 Hoare Triples 3

2.1.2 Assignment Axiom 3

2.1.3 Rule of Composition 4

2.2 OCaml 4

2.2.1 Immutability by default 5

2.2.2 ADTs (Algebraic Data Types) 6

2.2.3 First-Class and Higher-Order Functions 6

2.2.4 Modules, Functors and Signatures 6

2.2.5 Type Abstraction and Encapsulation 7

2.3 Standard ML 8

2.4 Why3 8

2.5 Cameleer 9

3 State Of The Art **10**

3.1 Certified Compilers 10

3.1.1 CompCert 10

3.1.2 CakeML 10

3.2 Pipeline 10

4 Preliminary Results **12**
4.1 Tool Modifications 12
4.2 Case Studies 12
4.2.1 Tail Recursive Factorial 13
4.2.2 Exception recursive search 14
4.2.3 High-order 16
4.2.4 Depth-search tree 16
4.2.5 Tree Comparison 17

5 Work Plan **19**
5.1 From OCaml to CakeML 19
5.1.1 Correct Boolean Literals Syntax 19
5.1.2 Correct Reference Declaration Syntax 19
5.1.3 Correct Let-Binding Syntax 19
5.1.4 Correct Array Operations Syntax 19
5.1.5 Correct Datatype Syntax 19
5.1.6 Correct Exception Handling 19
5.2 Translation from CakeML to OCaml 20
5.2.1 Arithmetic and Boolean operations 20
5.2.2 Let-Bindings 20
5.2.3 References 20
5.2.4 Array Operations 20
5.2.5 Datatypes 20
5.2.6 Correct Exception Handling 20
5.3 Dissertation 20
5.3.1 Writing 20

Bibliography **22**

List of Figures

3.1 OCaml to WhyML pipeline with Cameleer 10
3.2 Goal pipeline 11

5.1 Tentative Schedule 21

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. ii, 10–15, 18–20)*
Cameleer	An automated deductive verification tool for OCaml programs using GOSPEL annotations. *(pp. ii, v, 10–12)*
GOSPEL	A specification language for the OCaml language, intended to be used in various purposes. The acronym stands for Generic OCaml SPEcification Language. *(pp. ii, 6, 9–14, 16–18)*
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. ii, v, 5–7, 9–20)*
Why3	A platform for deductive program verification that relies on external provers. *(pp. ii, 4, 6, 8–10, 12–14)*
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. ii, v, 8, 10–12)*

1 Introduction

1.1 Motivation

Progress in deductive software verification has been steadily advancing, particularly for formal languages. However, the foundational groundwork was laid as early as 1936 by Alan Turing in his seminal paper "On Computable Numbers," which introduced key concepts of computation and formal proof. Other notable papers where mathematical propositions were established purely through theoretical reasoning include Alan Turing's Systems of Logic Based on Ordinals, Alonzo Church's An Unsolvable Problem of Elementary Number Theory and Kurt Gödel's Incompleteness Theorems. These works laid important foundations in logic, computation, and formal reasoning without relying on physical implementation or empirical methods. Our research draws on several key papers that directly influenced deductive software verification and the technologies we will use. These include Robin Milner's foundational work on type polymorphism (shaped the ML language) Gordon and Melham's Introduction to HOL (formalized higher-order logic for verification) and Xavier Leroy's CompCert (a landmark in formally verified compilation) [16, 17].

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 behavior, invalidating the higher-level correctness so compilers like CompCert and CakeML address this issue by providing formally verified compilation pipelines [11, 6, 10], thereby eliminating a major source of uncertainty and ensuring that correctness is preserved from source code to machine code [9].

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 [4]. 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, with providing fomal language specification tightly integrated with OCaml, verification has become easier with a modular specification that doesn't need much 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, Agda, F*(F star), Liquid Haskell and WhyML have paved the way. Now that we have a behavioural specification language for OCaml we can expand this verification for other funtional languages, that was already done in 2021 with the addition of Cameleer, an automated deductive verification tool that translates a formally-specified program, like GOSPEL, into the corresping code in WhyML. This innovation in translation of verified code to other languages gave a new view onto how other functional languages could have their code verified while being written in a more expressive language just like OCaml. A clear applicant was CakeML, a language based on a substantial subset of Standard ML, having a core goal of creating an end-to-end verified compiler.

1.2 Problem Definition

Writing precise specifications can turn out to be very challenging, since having an incomplete specification will eventually make the verification meaningless. PUT SOME RESEARCHES ABOUT SOFTWARE VERIFICATION AND TALK ABOUT HOW THERE ARE NO AUTOMATED VERIFICATION PAPERS. Despite all the papers above mentioned there are not much papers that go deep inside automated deductive verification. And then we have CakeML, a research-driven compiler with the main goal of providing a fully proof-producing code generation tool that given ML-like functions in higher-order logic (HOL) automatically produces equivalent executable machine code. Analyzing syntactic and semantic foundations with OCaml we see that both share very similar features most notably, functional core, strong static typing, pattern matching and higher-order functions. Now we are presented with some questions:

- Now that automated deductive verification has a tool that eases translation, could we expand it for even more languages? - Can CakeML's verified compilation pipeline be generalized to other ML-family languages like WhyML? - What minimal syntactic and semantic guarantees must a language offer to be compatible with CakeML's verified compiler? - Could an OCaml-to-HOL4 transpiler (guided by GOSPEL specs) be created to automate CakeML target generation?

1.3 Goals and Expected Contribution

1.4 Report Structure

2
Background

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, C.A.R. Hoare introduced a new logic 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.

"The purpose of this study is to provide a logical basis for proofs of the properties of a program" [7].

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 (fragment) and Q is a post-condition:

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

A Hoare triple expresses a partial correctness guarantee: if the precondition P holds before executing a program fragment C, and if C terminates, then the postcondition 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

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

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 postcondition P holds after assigning the expression f to the variable x, it suffices to prove the precondition 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 postcondition of the first program segment matches the precondition of the second, then the entire program will produce the intended result—assuming the initial precondition of the first segment holds.

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

Hoare Logic was significantly strengthened by Cook's seminal work on the soundness and completeness of an axiom system for program verification. In his paper, 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].

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 preconditions and invariants. To address this, Edsger W. Dijkstra introduced the weakest precondition calculus, which reformulates program correctness into a computational problem: given a command C and a desired postcondition Q, the function $wp(C, Q)$ computes the weakest precondition P such that $\{P\}C\{Q\}$ holds [3].

This transformation from proof obligations to a calculable precondition function represents a critical step toward automating program verification. Unlike Hoare's original formulation, which requires deductive reasoning to derive correctness properties, weakest precondition semantics allow for algorithmic generation of verification conditions, thereby enabling integration with automated theorem provers and SMT solvers.

The influence of weakest preconditions is particularly evident in modern deductive verification tools such as Why3 [1], and Dafny [8].

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.

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. 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.

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, 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 promotted 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 behavior and eliminates many common sources of verification complexity, such as aliasing and unintended side effects.

OCaml

```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. Such properties are essential for theorem proving and are readily leveraged by formal tools like GOSPEL, Coq, and Why3.

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 Modules, Functors and Signatures

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.

OCaml

```
module type StackSig = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a t
end

module Stack : StackSig = struct
  type 'a t = 'a list
  let empty = []
  let push x s = x :: s
  let pop = function [] -> [] | _ :: tl -> tl
end

module type ElemSig = sig type t end

module MakeStack (_ : ElemSig) : StackSig = Stack
```

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.

2.2.5 Type Abstraction and Encapsulation

One of OCaml's strengths lies in its support for abstract types via module signatures. This allows developers to hide implementation details and expose only essential operations, enabling verification at the level of observable behavior rather than internal representation.

OCaml

```
module Counter : sig
  type t
  val create : unit -> t
  val incr : t -> t
  val get : t -> int
end = struct
  type t = int
  let create () = 0
  let incr x = x + 1
  let get x = x
end
```

By hiding the concrete type `int`, this interface prevents misuse and allows formal specification to focus solely on observable behavior. In verification tools, this maps cleanly to abstract state machines and promotes reasoning over behavior instead of implementation details.

2.3 Standard ML

Standard ML is a functional programming language that fully embraces the expressiveness of mathematical functions. However, it was also shaped by practical programming needs, leading to the inclusion of imperative constructs and a robust exception handling system. The language supports modularity through an advanced system of parametric modules, designed to facilitate the structured development of large-scale software systems. Moreover, Standard ML is strongly and statically typed, and it was the first programming language to introduce a form of polymorphic type inference that combines strong type safety with considerable flexibility in programming style [12].

One of the most distinguishing features of Standard ML is its formal definition, which precisely specifies the language's static and dynamic semantics using structural operational semantics (SOS). This operational style makes the semantics especially suitable for mechanization in proof assistants like HOL, bridging the gap between theoretical definitions and executable verification. This made it one of the first programming languages to be fully defined in a mathematical sense, laying a strong foundation for formal verification frameworks and verified compilers such as CakeML [12, 14].

Even though the foundational formal definition of Standard ML had already been established, the ability to embed and reason about its semantics within a proof assistant like HOL marked a significant advance. This transformation from a descriptive, paper based semantics to one that is executable and machine-verifiable played a key role in laying the groundwork for end-to-end verified compilers [15].

2.4 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 behaviors 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. [5] 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 CVC4, 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. [1]

2.5 Cameleer

With the evolution of proof assistants becoming pivotal for industrial-size projects the need for. Despite all the advances in deductive verification and proof automation, little attention has been given to the family of functional languages. Let us consider, for instance, the OCaml language. It is well suited for verification, given its well-defined semantics, clear syntax, and state-of-the-art type system. Yet, the community still lacks an easy to use framework for the specification and verification of OCaml code.

Cameleer [13]

```
let f x = x + 1 (*@ res = f x ensures res = x + 1)                GOSPEL + OCaml
```

3 State Of The Art

3.1 Certified Compilers

3.1.1 CompCert

3.1.2 CakeML

3.2 Pipeline

Some parts of the verification pipeline have already been implemented or require only minor adjustments to meet their objectives. As we know, Cameleer provides translation of OCaml code annotated with GOSPEL specifications into WhyML:

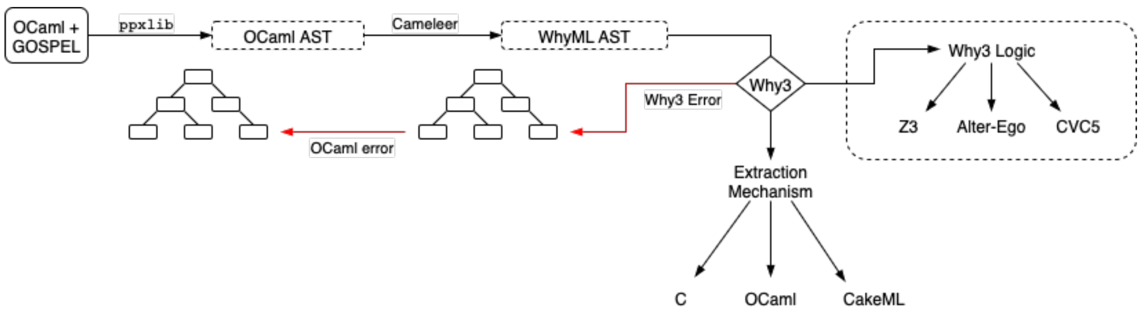

Figure 3.1: OCaml to WhyML pipeline with Cameleer

The extraction process does not require the correctness of the code to be proven. This may lead to potentially incorrect OCaml programs to be extracted to CakeML. Additionally, it is not designed to prevent users from extracting code even if the GOSPEL specifications can not guarantee correctness.

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 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. One of our goals is 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.

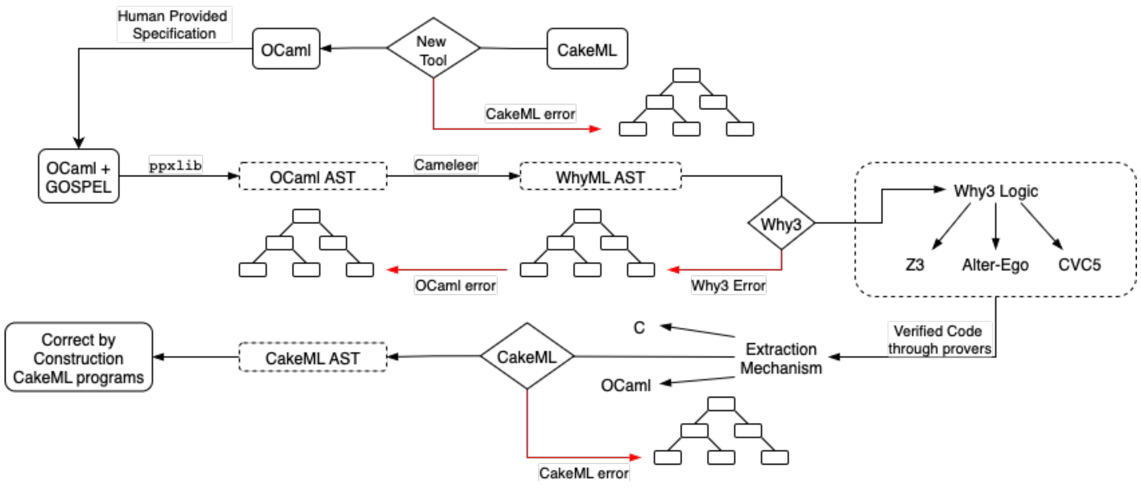

Figure 3.2: Goal pipeline

4

Preliminary Results

4.1 Tool Modifications

As shown in Figure 3.1, which outlines the Cameleer pipeline, there is no explicit support for CakeML. However, a small modification to the file `/bin/cli.ml` allows extending the existing extraction process. By changing the `execute_extract` function to invoke `why3 extract -D cakeml programa.mlw -o programa.cml`, the tool can target CakeML as a backend. With this adjustment, running `cameleer program.ml --extract` will translate 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 the flag `--debug` and then apply the mechanism of extraction of Why3 directly. Not only this process more complex, the debug flag produces code that doesn't let the launch of the provers from the Why3 ide because it contains proof goals directly in the code, which would imply also altering the code manually or to substantially modify the behaviour of the debug flag which is not intended.

4.2 Case Studies

For all of the case studies below mentioned, the OCaml + GOSPEL code was compiled and 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 will not be compilable. These issues reflect limitations in the current extraction process and highlight the need for further refinement.

4.2.1 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`. 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. The main function `fact` initializes the auxiliary call with appropriate base values. The GOSPEL annotations formally specify the expected behavior, including preconditions such as non-negativity of n, loop variants for termination, and the correctness of the returned result. This annotated version allows the function to be verified in Why3 before attempting any extraction.

GOSPEL + OCaml

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


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

From the code above, the translated code was this:

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. `let` bindings are not terminated explicitly, but in case of doubt it is possible to use the `begin...end` block. So for the correct writing of the above code there is only the need to add three `end` in function `fact_aux` and one `end` in function `fact`.

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
```

4.2.2 Exception recursive search

The next implementation showcases a recursive linear search on an array, utilizing exceptions to indicate when the search value is not found. This example allows us to examine the support provided by OCaml for data structures like arrays, as well as its mechanism for exception handling. The core function, `linear_search`, delegates the search process to an auxiliary function that recursively traverses the array.

The GOSPEL annotations specify the behavior formally by defining preconditions to ensure the array bounds are respected and that elements before the current index c have been verified not to match the target value n, because the search has not terminated yet. Postconditions guarantee that, if a value is returned, it is the searched value; and if an exception `Not_found` is raised, the value does not exist anywhere in the array. This makes the function verifiable by Why3 before attempting extraction to CakeML, ensuring that key correctness properties are formally validated.

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 n) 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 r = n
    variant Array.length a - c
*)

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 r = n
*)
```

Generated code from cameleer to CakeML.

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 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)` and automatically handles operations like `Array.length`, CakeML demands explicit and fully qualified references such as `Array.sub a i` and `Array.length a`. The generated code fails in this aspect by using invalid expressions such as `a1.length` and `get a1 c1`, neither of which are supported by the CakeML standard library or type system. This divergence in the array API further contributes to the failure of the generated code to compile.

Moreover, the syntax of let-bindings also contributes for another critical issue as mentioned above. These problems collectively illustrate the importance of a properly configured and semantically aware translation mechanism.

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
```

4.2.3 High-order

High-order logic code verified through cameleer.

GOSPEL + OCaml

```
    (*@ function rec map (f: int -> int) (l: int list) : int list =
            match l with
            | [] -> []
            | h::t -> (f h) :: map f t *)
    (*@ variant l
            ensures List.length result = List.length l *)

    let rec mult_list l n =
        match l with
        | [] -> []
        | h :: t -> n * h :: mult_list t n
    (*@ r = mult_list l n
            ensures r = map (fun x -> x * n) l
            variant l *)
```

Generated code from cameleer to CakeML.

CakeML

```
    fun mult_list l n = let val l1 = l in
        let val n1 = n in
        (case l1 of
            [] => []
        | h :: t => (n1 * h) :: (mult_list t n1))
```

Correct code that CakeML can compile

CakeML

```
    fun mult_list l n = let val l1 = l in
        let val n1 = n in
        (case l1 of
            [] => []
        | h :: t => (n1 * h) :: (mult_list t n1))
        end end
```

4.2.4 Depth-search tree

GOSPEL + OCaml

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

(*@ 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
*)

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
*)
```

Generated code from cameleer to CakeML.

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)))
```

Correct code that CakeML can compile

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
```

4.2.5 Tree Comparison

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
```

Generated code from cameleer to CakeML.

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)))
```

Correct code that CakeML can compile

CakeML

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

Without addressing 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 semantics of the original verified source.

5 Work Plan

5.1 From OCaml to CakeML

5.1.1 Correct Boolean Literals Syntax

Modifying boolean literals to match CakeML syntax, capitalizing initial letters (e.g., `true` → `True`).

5.1.2 Correct Reference Declaration Syntax

Adjusting reference declarations to follow CakeML conventions, including capitalization and syntax structure.

5.1.3 Correct Let-Binding Syntax

Ensuring all `let` bindings are properly closed with `end` as required by CakeML.

5.1.4 Correct Array Operations Syntax

Replacing OCaml array operations with their CakeML equivalents.

5.1.5 Correct Datatype Syntax

Fixing datatype definitions to respect CakeML ordering and syntax, especially for constructors with multiple arguments.

5.1.6 Correct Exception Handling

Refactoring exception declarations and usage to comply with CakeML, ensuring all exceptions are defined at the top level.

5.2 Translation from CakeML to OCaml

5.2.1 Arithmetic and Boolean operations

Mapping CakeML arithmetic and boolean operators to their OCaml equivalents, adjusting casing and syntax as needed.

5.2.2 Let-Bindings

Converting CakeML `let`... `end` structures into standard OCaml let-bindings without explicit termination.

5.2.3 References

Translating CakeML reference operations into OCaml.

5.2.4 Array Operations

Adapting CakeML array access and manipulation to OCaml syntax.

5.2.5 Datatypes

Reconstructing datatype definitions from CakeML into idiomatic OCaml type declarations, preserving constructors and structure.

5.2.6 Correct Exception Handling

Transforming exception declarations and usage from CakeML back into OCaml style with inline definitions and raises.

5.3 Dissertation

5.3.1 Writing

Finalizing the dissertation document by detailing the modifications made to the codebase, describing the implementation steps, and analysing the results and limitations of the translation process.

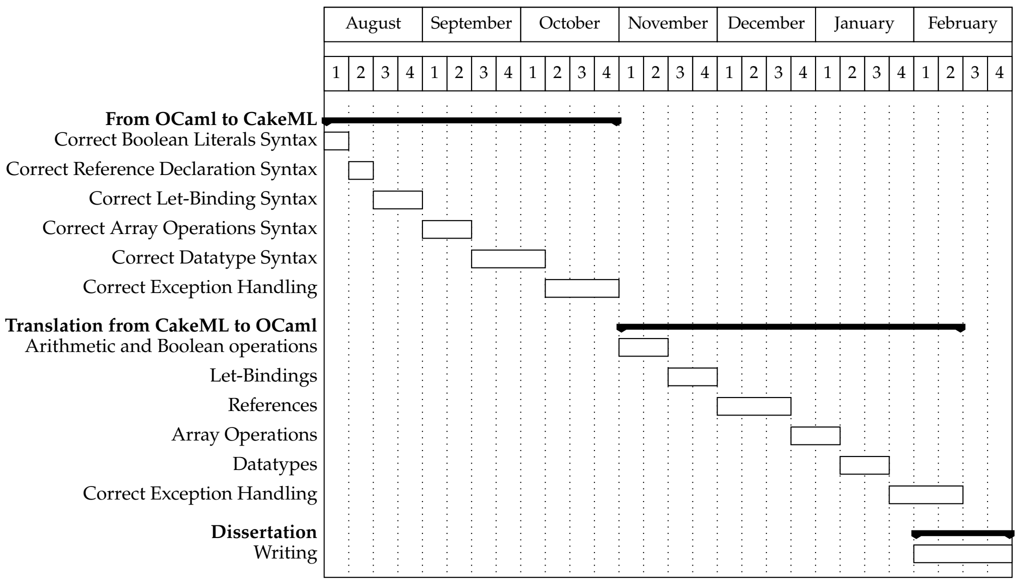

Figure 5.1: Tentative Schedule

Bibliography

[1] 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. 4, 9).

[2] 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).

[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://www.worldcat.org/oclc/01958445 (cit. on p. 4).

[4] 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 p. 1).

[5] 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 p. 8).

[6] 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).

[7] 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 p. 3).

[8] 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. 4).

[9] 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).

[10] 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 p. 1).

[11] 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).

[12] R. Milner. *The Definition of Standard ML: Revised*. Mit Press. Penguin Random House LLC, 1997. ISBN: 9780262631815. URL: https://books.google.pt/books?id=e0PhKfbj-p8C (cit. on p. 8).

[13] 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 p. 9).

[14] T. Sewell et al. "Cakes That Bake Cakes: Dynamic Computation in CakeML". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1121–1144. DOI: 10.1145/3591266. URL: https://doi.org/10.1145/3591266 (cit. on p. 8).

[15] D. Syme. "Reasoning with the Formal Definition of Standard ML in HOL". In: *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings*. Ed. by J. J. Joyce and C. H. Seger. Vol. 780. Lecture Notes in Computer Science. Springer, 1993, pp. 43–60. DOI: 10.1007/3-540-57826-9_124. URL: https://doi.org/10.1007/3-540-57826-9%5C_124 (cit. on p. 8).

[16] 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).

[17] 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).