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 9, 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 9, 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Currently, the extraction mechanism in Why3 to CakeML is outdated and only supports a subset of the language.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

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, 12–18, 20–24, 26, 28–30)*

Cameleer An automated deductive verification tool for OCaml programs using GOSPEL annotations. *(pp. ii, v, 9–14)*

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, 9–17, 19–23, 25, 26)*

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, 9–23, 25, 26, 28–30)*

Why3 A platform for deductive program verification that relies on external provers. *(pp. ii, v, 4, 6, 8–12, 14, 16, 18, 28)*

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, 12–14)*

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) [21, 22].

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 [13, 7, 12], thereby eliminating a major source of uncertainty and ensuring that correctness is preserved from source code to machine code [10].

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 [5]. 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" [8].

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

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

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 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 [3].

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 [4].

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 [9].

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 [14].

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 [14, 19].

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 [20].

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. [6] 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 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 [17]. 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 [2]. 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 [18, 16]. 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 [2].

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.

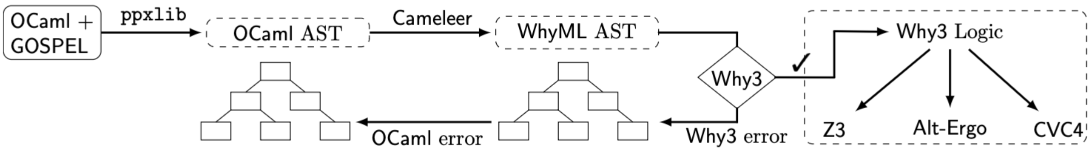

Figure 2.1: The Original Cameleer Pipeline [17]

The pipeline 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 [5].

Lets take this simple example for an iterative factorial in OCaml + GOSPEL after applying the Cameleer tool:

GOSPEL + OCaml

```
(*@
  function rec factorial (n:int) :int =
    if n=0 then 1 else n * factorial (n-1)
*)
(*@
  requires n >= 0
  variant n
*)
let fact n =
  let r = ref 1 in
    for i=1 to n do
      (*@invariant !r = factorial (i-1)*)
      r := !r*i
    done;
    !r
(*@
  r = fact n
  requires n >= 0
  ensures r = factorial n
*)
```

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:

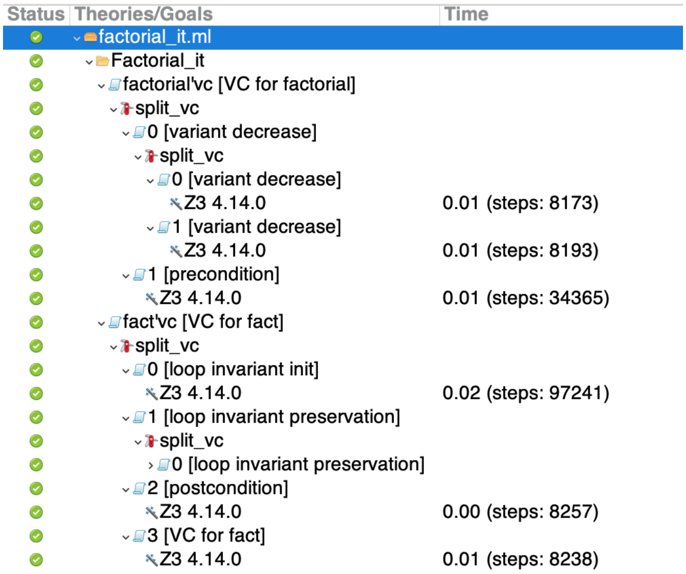

Figure 2.2: The Goals Proved in Why3

All verification goals were successfully discharged by 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

3.1 Certified Compilers

3.1.1 CompCert

[15, 11]

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:

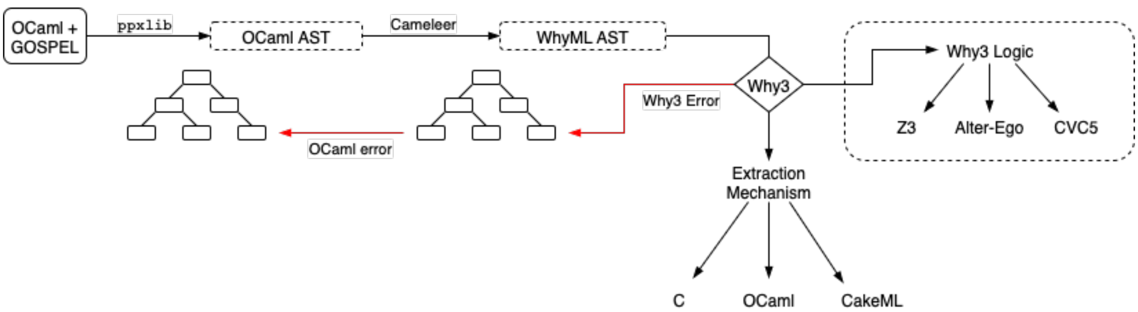

Figure 3.1: OCaml to WhyML pipeline with Cameleer

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.

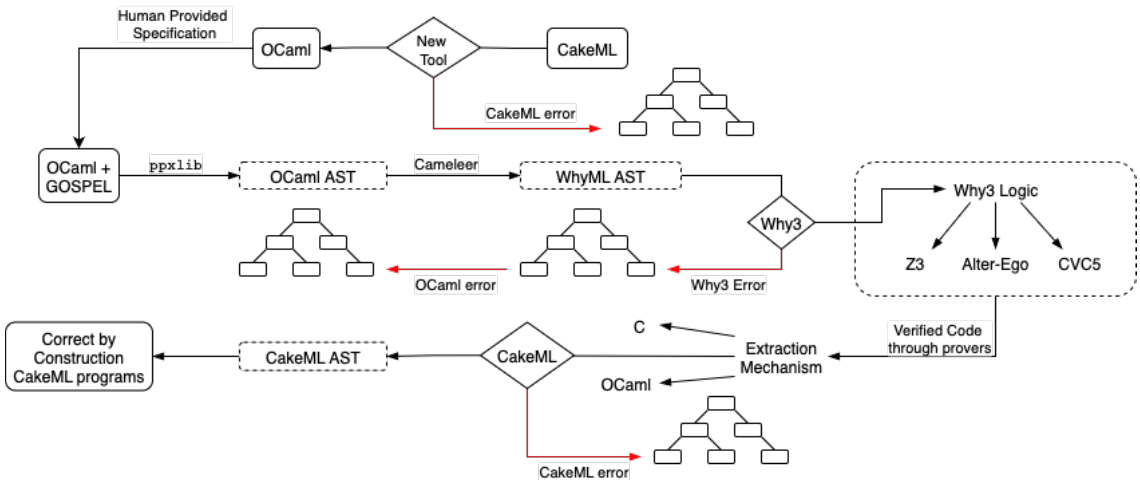

Figure 3.2: Goal pipeline

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 back-end. 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 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 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 may 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` 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 have some 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.

GOSPEL + OCaml

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

The main function `fact` 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.

4.2.2 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
  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 on each iteration of the array's length. 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 can not be the desired value, otherwise the function should have terminated earlier. Since the exception `Not_Found` was not caught this means the function only returns a value if this exception was not raised, therefore we only 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.

4.2.3 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 other types of 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 generate 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 the 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 operation is used as if it is an operation from the standard library, however CakeML doesn't have that library with that syntax. The let-bindings continue to generate for every argument with no termination clause.

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

4.2.4 High-order

In the documentation for CakeML it was not possible to find any mention of higher-order functions as such we devised an example that combines higher order with pattern matching and lists. The function in this example multiplies all the values of a list by a certain integer:

GOSPEL + OCaml
```
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 *)
```

This recursive function `mult_list` receives as arguments a list and a value that is used to multiply on each of the lists values. While using a pattern matching the function must continue until the original list is empty.

The original list has its values extracted one by one until the list is empty, signalling the function termination, which in turn represents the variant clause. The function `map` is classical example of function programming that applies another function to all elements of a list. One can observe that the `mult_list` function is a particular case of the `map` function, where the function is the multiplication by a given integer n and is expressed by an anonymous function.

The logical implementation of the `map` function is presented below:

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 *)
```

The logical function `map` represents the application of a certain function to the values of the list provided. As the same reason that variant clause is represented by the list, since the termination for this logical function is also determined by when the list is empty. The is also the need to ensure that the length of the list resulting from applying the function to the original list is the same since the number of values do not change with this transformation.

The corresponding generated code is:

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))
```

We can observe that pattern matching and basic list handling seems to be working well after the translation, however the extraction still generates clones of the arguments, the code can not be compilable.

By solving the issues found, we tested the code and effectively, high-order functions are allowed by the compiler in CakeML. The corrected code is displayed below:

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
```

Since the issue continues to stem from the let-bindings, the solution to add the termination token or eliminating the unnecessary let-bindings still seems to solve the problem at hand.

4.2.5 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 reason, 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.

4.2.6 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
```

The function `cmp` uses pattern matching to compare the trees, in case both are leaves it is true, if one is a leaf, but the other is not they are different, finally if both are nodes then it depends on the respective root and subtrees.

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.

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, searching on the how-to document from CakeML we could find how modules were constructed and used that to create the structure `Tree` with the correct syntax, which was not present in the translated code.

4.2.7 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 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 equals 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 through 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.

4.3 Analyses For The Case Studies

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 behaviour of the original verified source.

5 Work Plan

5.1 From OCaml to CakeML

5.1.1 Correct Boolean Literals Syntax

In CakeML, the 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`).

5.1.2 Correct Reference Declaration Syntax

Just as the boolean literals, references also start with uppercase which is also not the same as OCaml. The same capitalization solution will be applied for reference declarations in order to follow CakeML conventions (e.g., `ref` → `Ref`).

5.1.3 Correct Let-Binding Syntax

Regarding the let-binding structure in CakeML, this alwasy 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-binding, this is redundant, and it is our objective to simplify the resulting translations.

5.1.4 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.

5.1.5 Correct Data Type Syntax

When defining the data types, the syntax from OCaml and CakeML have some differences when it comes to tuple declarations, in particular, the `of` keyword and the separator `*` found in OCaml are not used in CakeML. However, the translation mechanism misplaces the generic type before the `datatype` keyword in which the data type is written. The solution comes through fixing data type definitions to respect CakeML ordering and syntax.

5.1.6 Correct Exception Handling

In OCaml we can declare exceptions within functions, which is not possible in 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

5.2.1 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 adjusting casing and syntax will be performed.

5.2.2 Let-Bindings

The next for the translation tool is to support let-bindings. 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 lets are nested due to the environments of the bindings.

5.2.3 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.

5.2.4 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.

5.2.5 Data Types

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

5.2.6 Correct 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 a 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

5.3.1 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

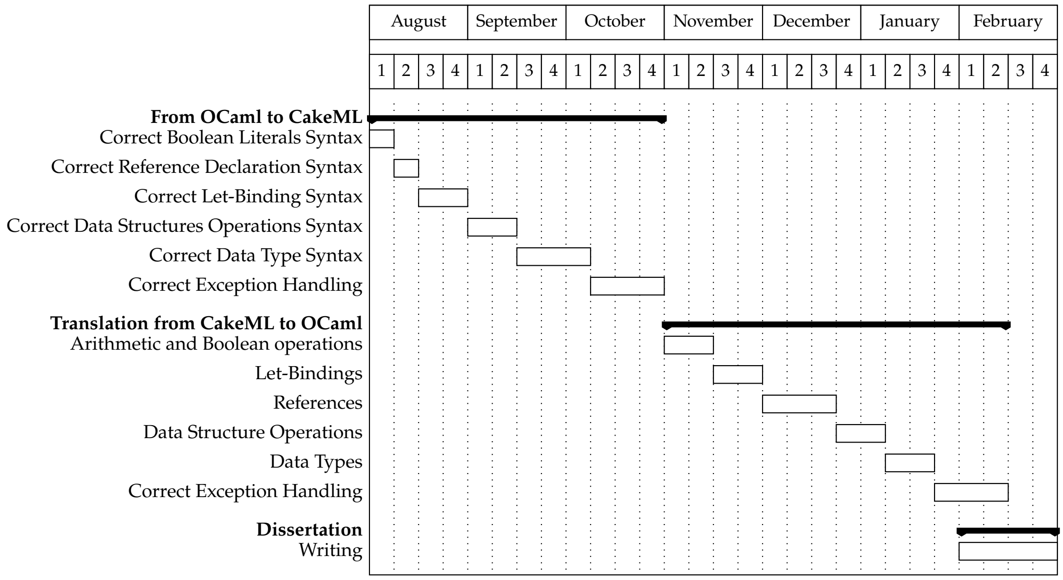

Figure 5.1: Tentative Schedule

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] 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 p. 9).

[3] 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).

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

[5] 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, 10).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] 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 p. 12).

[12] 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).

[13] 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).

[14] 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).

[15] 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. 12).

[16] 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. 9).

[17] 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. 9, 10).

[18] 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. 9).

[19] 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).

[20] 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).

[21] 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).

[22] 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).