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: *June 29, 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: *June 29, 2025*

Abstract

Regardless of the language in which the dissertation is written, usually there are at least two abstracts: one abstract in the same language as the main text, and another abstract in some other language.

The abstracts' order varies with the school. If your school has specific regulations concerning the abstracts' order, the NOVAthesis LaTeX (novathesis) (LaTeX) template will respect them. Otherwise, the default rule in the novathesis template is to have in first place the abstract in *the same language as main text*, and then the abstract in *the other language*. For example, if the dissertation is written in Portuguese, the abstracts' order will be first Portuguese and then English, followed by the main text in Portuguese. If the dissertation is written in English, the abstracts' order will be first English and then Portuguese, followed by the main text in English. However, this order can be customized by adding one of the following to the file `5_packages.tex`.

```
\ntsetup{abstractorder={<LANG_1>,...,<LANG_N>}}
\ntsetup{abstractorder={<MAIN_LANG>={<LANG_1>,...,<LANG_N>}}}
```

For example, for a main document written in German with abstracts written in German, English and Italian (by this order) use:

```
\ntsetup{abstractorder={de={de,en,it}}}
```

Concerning its contents, the abstracts should not exceed one page and may answer the following questions (it is essential to adapt to the usual practices of your scientific area):

1. What is the problem?
2. Why is this problem interesting/challenging?
3. What is the proposed approach/solution/contribution?
4. What results (implications/consequences) from the solution?

Keywords: One keyword, Another keyword, Yet another keyword, One keyword more, The last keyword

Resumo

Independentemente da língua em que a dissertação está escrita, geralmente esta contém pelo menos dois resumos: um resumo na mesma língua do texto principal e outro resumo numa outra língua.

A ordem dos resumos varia de acordo com a escola. Se a sua escola tiver regulamentos específicos sobre a ordem dos resumos, o template (LaTeX) novathesis irá respeitá-los. Caso contrário, a regra padrão no template novathesis é ter em primeiro lugar o resumo *no mesmo idioma do texto principal* e depois o resumo *no outro idioma*. Por exemplo, se a dissertação for escrita em português, a ordem dos resumos será primeiro o português e depois o inglês, seguido do texto principal em português. Se a dissertação for escrita em inglês, a ordem dos resumos será primeiro em inglês e depois em português, seguida do texto principal em inglês. No entanto, esse pedido pode ser personalizado adicionando um dos seguintes ao arquivo `5_packages.tex`.

```
\abstractorder(<MAIN_LANG>):={<LANG_1>,...,<LANG_N>}
```

Por exemplo, para um documento escrito em Alemão com resumos em Alemão, Inglês e Italiano (por esta ordem), pode usar-se:

```
\ntsetup{abstractorder={de={de,en,it}}}
```

Relativamente ao seu conteúdo, os resumos não devem ultrapassar uma página e frequentemente tentam responder às seguintes questões (é imprescindível a adaptação às práticas habituais da sua área científica):

1. Qual é o problema?
2. Porque é que é um problema interessante/desafiante?
3. Qual é a proposta de abordagem/solução?
4. Quais são as consequências/resultados da solução proposta?

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

Contents

List of Figures

Glossary

GOSPEL A specification language for the OCaml language, intended to be used in various purposes. The acronym stands for Generic OCaml SPEcification Language. *(p. 4)*

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. *(p. 4)*

Acronyms

novathesis NOVAthesis LaTeX *(pp. i, ii)*

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).

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 [**<empty citation>**]. 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.

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, thereby eliminating a major source of uncertainty and ensuring that correctness is preserved from source code to machine code.

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.

Hoare Logic, introduced by C.A.R. Hoare in his seminal 1969 paper, is a formal system designed to reason about the partial correctness of computer programs. Inspired by mathematical axioms and formal proof methods, it uses assertions and inference rules to specify and verify program behavior. Hoare's framework laid the foundation for systematic program verification and highlighted the importance of modeling low-level computational constraints, such as finite precision, overflows, and machine-specific arithmetic within formal systems.

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

Assignment Axiom

$$\{P_0\}x := f\{P\}$$

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.

Rule of Composition

$$\vdash P\{Q_1\}R_1 \quad \text{and} \quad \vdash R_1\{Q_2\}R \quad \text{then} \quad \vdash P\{(Q_1;Q_2)\}R$$

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.

2.2 OCaml

2.3 Standard ML

2.4 Why3

2.5 Cameleer

Cameleer [1]

```
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

4

Preliminary Results

5

Work Plan

Bibliography

[1] 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. 4).