JOÃO FRANCISCO SERRENHO NINI
BSc in Computer Science

FORMAL VERIFICATION OF PROGRAMS EQUIVALENCE

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

DEPARTMENT OF
COMPUTER SCIENCE

FORMAL VERIFICATION OF PROGRAMS EQUIVALENCE

JOÃO FRANCISCO SERRENHO NINI

BSc in Computer Science

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

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

Abstract

Ensuring that a program functions as intended is a complex issue that has been approached many times and in various ways throughout history. Human-assisted software verification is the most complete and reliable method, despite its inherent additional effort.

The objective of this work is to ease proofs of complex programs by taking advantage of a relation to an equivalent program that is easier to prove. If it is possible to establish that two different programs are equivalent, it is also very likely that reusing the simplest specification will lead to a faster and easier proof of the more sophisticated program. Relational Hoare Logic paved the way to the development of several techniques to reason about the similarities of two different programs. In this work, we will base our approach on the concept of *product programs* to reduce relational verification into standard verification.

Furthermore, there will be a description of the relevant background in order to effectively comprehend the approach we chose, as well as a presentation of the other possible methods to achieve what we propose. Finally, we will demonstrate a few good initial results obtained by coding and specifying two different implementations of the same algorithm, using different paradigms.

Keywords: Deductive verification, Program equivalence, OCaml, Cameleer, Relational Hoare logic, Product programs

Resumo

Garantir que um programa se comporta como esperado é um assunto complexo que tem sido alvo de muitas e diferentes tentativas ao longo da história. A verificação de software assistida por humanos é o método mais confiável e completo, apesar do seu inerente esforço adicional.

O objetivo deste trabalho é facilitar as provas de programas complexos a partir da relação entre este e um equivalente que seja mais simples de provar. Se for possível estabelecer que dois programas diferentes são equivalentes, então é altamente provável que reutilizar a especificação do mais simples levará a uma prova mais fácil e rápida do programa mais sofisticado. A lógica de Hoare relacional abriu caminho para o desenvolvimento de várias técnicas para raciocinar sobre as similaridades de dois programas diferentes. Neste trabalho, vamos basear a nossa abordagem no conceito de *product programs* para reduzir a verificação relacional a verificação "tradicional".

Além disso, haverá uma descrição do conhecimento prévio necessário para compreender efetivamente a nossa abordagem, tal como a apresentação de outros possíveis métodos para alcançar aquilo a que nos propomos. Finalmente, demonstraremos alguns resultados iniciais positivos obtidos pela escrita do código e especificação de duas implementações diferentes do mesmo algoritmo, utilizando paradigmas diferentes.

Palavras-chave: Verificação dedutiva, Equivalência de programas, OCaml, Cameleer, Lógica de Hoare relacional, Product programs

Contents

List of Figures

1 Introduction

1.1 Motivation

The impact of errors in programs can vary greatly. Sometimes it can make you lose a match of a game you play to relax after a long, stressful day. But sometimes it can make commercial airplanes with hundreds of people catastrophically crash. However, there are other situations as dangerous as a plane crash: medical equipment failure, nuclear accidents, defense systems errors... the list is not short. Clearly, the approach to ensure the correctness of some software has to be extremely thorough and methodical.

Deductive verification has proven to be the most accurate way to prove that a given program outputs what is intended given its inputs. Although this is the most effective way, it is also the one that requires the most investment: professionals that can write the specifications for the programs and then help theorem solver discard verification conditions. As the reader can imagine, most companies prefer to resort to the cheaper, faster methods of testing. Therefore, there has to be a simpler way of proving, still using deductive verification, that a program is correct. That way is program equivalence and product programs.

1.2 Problem Definition

The question that this work focus its attention on is:

Can we automatically prove that two programs P_1 and P_2 are correct and equivalent?

Despite looking like a simple question, there is no simple answer. This is an issue that people have tried to address for decades, yet we did not find a definitive solution. There are several schools of software verification [1] but, as expected, all of them have advantages and disadvantages. However, we believe that human-assisted is the way, since it leaves many less bugs behind, compared to under-approximate methods and, at the same time, does not alert the programmer to errors that are not actually errors. Of course that this carries an extra human effort, in developing the specification for the programs and sometimes in discharging verification conditions.

In the context of human-assisted verification and programs P_1 and P_2, we need to first prove that the simpler program is correct, using already existent methods and then establish a connection between them to prove that the more complex is also correct. Product programs appear as a promising solution that came to light from the combination of two other methods called self-composition and cross-products. This solution takes the best of both worlds in order to overcome their limitations, which will be explained later, in chapter 3. Our work will, therefore, adapt product programs to be applied in the OCaml programming language, which could then be verified using Cameleer.

1.3 Goals and Expected Contributions

This work aims to bring to OCaml the concept of *product programs*, facilitating the task of proving that two different programs are behaviorally equivalent. This is especially relevant in cases where proving a simpler program and the relation between that and a more complex version is more peaceful than proving the correctness of both versions separately.

1.4 Report Structure

- Chapter 2 contains a series of concepts, like relational Hoare logic, and tools, such as Cameleer, that are essential to understand the foundations of our work.
- In chapter 3, we will enumerate and explain the most relevant developments regarding the subject of program equivalence proof.
- Chapter 4 contains the current state of our practical results, more specifically, two proofs of equivalence of simple OCaml programs, using Cameleer.
- Finally, chapter 5 describes what will be our tasks, when they will start and how long they will take.

2 Background

2.1 Formal Software Verification

The quest for proving that a program does what we expect it to do is so important in computer science that it became a branch of it called formal software verification. There are several approaches [1] that try to solve this issue, each one with different characteristics, advantages and disadvantages. In this section, we will describe the current situation of the theoretical work and tools that constitute the background.

2.1.1 Hoare Logic

Hoare logic [2] is a way of reasoning about the correctness of a program, following well defined axioms and rules. The main feature of the Hoare logic is the Hoare triple. The name comes from the 3 parts that constitute it: the pre-condition (P), the program (S) and the post-condition (Q):

$$\{P\}\ \mathbf{S}\ \{Q\}$$

This means that if a program S satisfies a given pre-condition P, the state of the program after the execution finishes satisfies the post-condition Q. To simplify a lot, you can look at the triple as a simple logical consequence $P \Rightarrow Q$. In the same way as in the logical consequence, if P is false, we can not say anything certain about the value of Q (after the execution terminates, in the case of the triple).

Let us now look at an example of the usage of the Hoare triple. Consider the following program:

$$S \triangleq \textbf{if}\ \text{grade} \geq 10\ \textbf{then}\ \text{reward} := 10\ \textbf{else}\ \text{reward} := 0$$

Hoare logic can be used to prove this triple:

$$\{\text{grade} \geq 0 \land \text{grade} \leq 20\}\ \mathbf{S}\ \{(\text{grade} \geq 10 \land \text{reward} = 10) \lor (\text{grade} < 10 \land \text{reward} = 0)\}$$

This triple describes a situation where if the value of *grade* is "valid" (considering grades from 0 to 20), we can say that the value of *reward* will be either 0 or 10, depending if the value of *grade* was less than 10 / greater or equal to 10, respectively. It is also important to mention the difference between *partial correctness* and *total correctness*. If *total correctness* is what you are looking for, you need to guarantee that the execution of the program finishes. Otherwise, you can only guarantee *partial correctness*.

2.1.2 Relational Hoare Logic

2.1.2.1 Evolution from Classical[1] Hoare Logic

Relational Hoare Logic [3, 4] (RHL) is a way of reasoning about some property of two different programs or two different executions of the same program. Some of these properties are observational equivalence and 2-properties, being non-interference and continuity two examples of that concept. While classical HL reasons about the satisfaction of the post-condition if the pre-condition is satisfied, RHL analyzes the relation between two programs in terms of their corresponding evolution from a given pre-condition to a post-condition. In other words, if programs *P1* and *P2* both respect the pre-condition (let us name it Φ), they will end (if their executions terminate) both respecting the post-condition (let us name it Ψ) or none of them will. This means that one of the programs satisfying Ψ and the other not satisfying it is not valid in RHL. This idea is known as the Hoare quadruple:

$$\{\Phi\}\, P1 \sim P2\, \{\Psi\}$$

So, why was RHL created when we already had classical HL? The main reason lies in the fact that the classical version does not have a direct or easy way of specifying the relation between two programs. This would clearly be an obstacle to this work since we aim to prove the correctness of programs by establishing a relation between them and their already proven or versions that are easier to prove. However, this work is definitely not the only one that relies on the connection between two different implementations of the same of code. The most common examples are probably compiler optimizations. Nonetheless, there is another example: when a given program has different algorithms that perform the same task and they get chosen depending on some characteristics that make one implementation more performant than some other(s).

2.1.2.2 Detailing RHL with examples

Next, there are some requirements not specified previously when presenting the Hoare quadruple. These are important details that have to be accounted for and will be described using example programs to make the explanation more tangible. Consider the notation *Pi.j* that will be used in this subsubsection, where *i* represents a given program index, *Pi* refers to a given program and *Pi.j* the identifier of a variable *j* in program *Pi*.

The first example corresponds to two different programs, let us call them *P1* and *P2*, and they both contain only two assignment statements applied to the same group of variables (*a, b, c*):

[1] When we say Classical Hoare Logic, we mean the original Hoare logic developed by Tony Hoare.

```
P1            P2
b := a + 1;   c := a + 2;
c := b + 1;   b := c - 1;
```

We can see that *b* and *c* do not change the final state of any of the programs, so, to write a Hoare quadruple that creates an equivalence between *P1* and *P2*, we need the pre-condition to assert that *P1.a* = *P2.a*:

$$\{\Phi\}\, P1 \sim P2\, \{\Psi\}$$

$$where\ \ \Phi \triangleq \text{P1.a} = \text{P2.a}$$
$$and\ \ \Psi \triangleq \text{P1.a} = \text{P2.a} \wedge \text{P1.b} = \text{P2.b} \wedge \text{P1.c} = \text{P2.c}$$

This Hoare quadruple should be read as: *If an execution of P1 and P2 guarantee the equality of their x-components, they both diverge from the post-condition or they both finish in states where their x-components, y-components and z-components have the same values* [2].

We now introduce the concept of separability. If two programs *P1* and *P2* utilize disjoint sets of variables, we say that they are *separable*. Also, in those cases we do not need to specify the program that a given variable belongs to when describing the pre and post conditions. Below is another example to show the impact on conciseness that this creates. Consider two programs, *P1*, whose variables are {a, b, i, j} and *P2* whose variables are {a', b', i', j'}, meaning that they are separable:

```
P1                P2
a := 1;           a' := 1;
while a < b do    i' := j' * j';
   i := j * j;    while a' < b' do
   a := a * i;       a' := a' * i';
od                od
```

What these programs calculate is not the focus here but notice a subtle, yet relevant difference: *i' := j' * j';* is out of the loop in *P2*. This is a common compiler optimization called *invariant hoisting*. *Why would you waste time performing a task several times if you can do it only once and get the same output?* In this example, the Hoare quadruple would be:

$$\{\Phi\}\, P1 \sim P2\, \{\Psi\}$$

$$\text{where }\ \Phi \triangleq (b = b') \wedge (j = j')$$
$$\text{and }\ \Psi \triangleq (a = a') \wedge (b = b') \wedge (i = i') \wedge (j = j')$$

Since the previous values in *a* and *i* (respectively *a'* and *i'*) are discarded by the assignments made to these variables, they will have no effect on the final state, so they do not appear in the pre-condition. On the other hand, to make sure that the components of

[2] The values of the components of any of the programs can differ between themselves (*P1.x* has no relation to *P1.y*, for example).

P1 and *P2* hold the same values after the executions terminate, we must assert that the n-components and y-components are equal in the initial state of the programs.

Furthermore, there is a special case of the usage of the Hoare quadruple that does not challenge any concept described before. Usually, in the context of the Hoare quadruple, we consider that the programs *P1* and *P2* are different, but what would happen if they are exactly the same? Well, since they are equal letter for letter, it is natural that they will always output the same results given the same inputs.

Finally, consider again two equal programs *P1* and *P2*. The way one should look at the transformation of Hoare triples into Hoare quadruples is that if $\{\Phi\}$ *P1* $\sim$ *P2* $\{\Psi\}$ is true, then $\{\phi\}$ *P1* $\{\psi\}$ will be true as well.

2.1.2.3 Axioms and Inference Rules

Consider, for the rules in this subsubsection, that the two given programs *C1* and *C2* are separable. To derive valid Hoare quadruples, there are axioms and inference rules that reason about the relation between two programs. We can divide these rules in at least three sets that increment the former set with some additional rules, meaning there is the simplest set, the base set and the extended set. The simplest set has limitations that we will discuss after presenting them, but is the best starting point:

$$\frac{}{\vdash \{\Phi\}\ \textbf{skip} \sim \textbf{skip}\ \{\Phi\}}\ \textsc{skip}$$

$$\frac{}{\vdash \{\Psi[x_1 \mapsto E_1][x_2 \mapsto E_2]\}\ x_1 := E_1 \sim x_2 := E_2\ \{\Psi\}}\ \textsc{assignment}$$

$$\frac{\vdash \{\Phi\}\ C_1 \sim C_2\ \{\Theta\} \qquad \vdash \{\Theta\}\ C_1' \sim C_2'\ \{\Psi\}}{\vdash \{\Phi\}\ C_1; C_1' \sim C_2; C_2'\ \{\Psi\}}\ \textsc{sequencing}$$

$$\frac{\begin{array}{c}\vDash \Phi \rightarrow (B_1 = B_2)\\ \vdash \{\Phi \wedge B_1\}\ C_1 \sim C_2\ \{\Psi\}\\ \vdash \{\Phi \wedge \neg B_1\}\ C_1' \sim C_2'\ \{\Psi\}\end{array}}{\vdash \{\Phi\}\ \textbf{if}\ B_1\ \textbf{then}\ C_1\ \textbf{else}\ C_1'\ \textbf{fi} \sim \textbf{if}\ B_2\ \textbf{then}\ C_2\ \textbf{else}\ C_2'\ \textbf{fi}\ \{\Psi\}}\ \textsc{conditional}$$

$$\frac{\vdash \{\Phi \wedge B_1\}\ C_1 \sim C_2\ \{\Phi\} \qquad \vDash \Phi \rightarrow (B_1 = B_2)}{\vdash \{\Phi\}\ \textbf{while}\ B_1\ \textbf{do}\ C_1\ \textbf{od} \sim \textbf{while}\ B_2\ \textbf{do}\ C_2\ \textbf{od}\ \{\Phi\}}\ \textsc{while}$$

$$\frac{\vDash \Phi' \rightarrow \Phi \qquad \vdash \{\Phi\}\ C_1 \sim C_2\ \{\Psi\} \qquad \vDash \Psi \rightarrow \Psi'}{\vdash \{\Phi'\}\ C_1 \sim C_2\ \{\Psi'\}}\ \textsc{weakening}$$

Figure 2.1: Simplest set of rules of RHL.

This set of rules has some considerable limitations however: both programs *C1* and *C2* must be structurally equivalent and execute in lockstep. This creates a situation where we can not derive a simple Hoare quadruple like {Φ} *C*; **skip** ~ *C* {Ψ}. To allow this and other important derivations, we present next other 4 rules that enable the separate reasoning of *C1* and *C2*. Together with the rules of the simplest set, these rules constitute the base set:

$$\frac{\vdash \{\Phi[x \mapsto E]\}\ \mathbf{skip} \sim C\ \{\Psi\}}{\vdash \{\Phi\}\ x := E \sim C\ \{\Psi\}}\ \textsc{assignment-L}$$

$$\frac{\vdash \{\Phi[x \mapsto E]\}\ C \sim \mathbf{skip}\ \{\Psi\}}{\vdash \{\Phi\}\ C \sim x := E\ \{\Psi\}}\ \textsc{assignment-R}$$

$$\frac{\vdash \{\Phi \wedge B\}\ C_1 \sim C\ \{\Psi\} \qquad \vdash \{\Phi \wedge \neg B\}\ C_2 \sim C\ \{\Psi\}}{\vdash \{\Phi\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi} \sim C\ \{\Psi\}}\ \textsc{conditional-L}$$

$$\frac{\vdash \{\Phi \wedge B\}\ C \sim C_1\ \{\Psi\} \qquad \vdash \{\Phi \wedge \neg B\}\ C \sim C_2\ \{\Psi\}}{\vdash \{\Phi\}\ C \sim \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ \{\Psi\}}\ \textsc{conditional-R}$$

Figure 2.2: Base set of rules of RHL.

Finally, the extended set contains one more rule than the base set: self-composition. We will not dive deep here on the need of this rule, instead we do it in this section.

$$\frac{\vdash \{\Phi\}\ C_1; C_2\ \{\Psi\}}{\vdash \{\Phi\}\ C_1 \sim C_2\ \{\Psi\}}\ \textsc{self-composition}$$

Figure 2.3: Extended set of rules of RHL.

Note that the premise of this rule is not a Hoare quadruple but a Hoare triple since there is no ~ operator, only a sequence of *C1* and *C2* within a single program. This triple can also be presented as the following quadruple:

$$\{\Phi \wedge \Phi'\}\ C1\ ;\ C2\ \sim\ C1'\ ;\ C2'\ \{\Psi \wedge \Psi'\}$$

If we keep the programs *C1;C2* and *C1';C2' separable* by renaming their variables and if $\Phi = \Phi'$ and $\Psi = \Psi'$, we get to the self-composition rule in the figure.

2.2 OCaml

2.2.1 History & Characteristics

OCaml [5] is a general purpose programming language that was released in 1996 at the National Institute for Research in Digital Science and Technology (Inria). It is usually seen as an extension of Caml (a dialect of the Meta Language) that includes object-oriented features, making OCaml a very versatile language. Besides supporting the functional, imperative and object-oriented approaches, it is also a language that can be either interpreted or compiled, to either bytecode or native code. Regarding its type system, OCaml comes with a strong and static type system that also features type inference.

Although the imperative programming paradigm is still more popular than its functional counterpart, the latter has been conquering space inside the universe of programming languages whose focus is on the imperative/object-oriented approach; for example, Java and Kotlin. Java received its first functional features in version 1.8 [6], released in 2014 and they have been expanding since. Kotlin, firstly released in 2016, was already designed with the functional style in mind [7].

2.2.2 Relevance

The relevance of OCaml and functional programming languages in general is an important question that I have asked myself several times in the past, and not only because the first contact with this paradigm was challenging. Although I do not question the relevance of these languages anymore because of the evidence that I will show next, I still think that it is not an unexpected concern amongst computer science students. I am mentioning this because the majority of jobs and businesses in the field require people for front-end development (with Javascript), back-end development (mostly with imperative languages) or, more recently, AI-related work (mostly with Python or other science-oriented languages). There is, although, strong evidence that OCaml is far from being an "academia only" programming language. OCaml is used in tech giants such as Meta and Microsoft, in Bloomberg L.P. (that even created an OCaml to Javascript compiler backend), in popular tools like Docker and in many other companies [8]. But there is more.

Here are a few impressive examples of what this programming language has been used to build:

1. Alt-Ergo [9], a popular SMT solver that has presence in Cameleer;
2. Coq [10], a formal proof management system;
3. and even the web version of Facebook Messenger [11]!

2.2.3 Code Examples

In this subsection, we will look at a simple Ocaml code example that determines the greatest common divisor of two given numbers, *a* and *b*. The first implementation represents the use of the functional paradigm while the second demonstrates how to use imperative constructs in this language.

OCaml

```
let rec gcd (a: int) (b:int) : int =
    if b = 0 then a
    else gcd b (mod a b)
```

```
let gcd_iter (a0: int) (b0: int) : int =                                    OCaml
    let b = ref b0 in
    let a = ref a0 in
    while !b ≠ 0 do
        let tmp = !a in
        a := !b;
        b := tmp mod !b
    done;
    !a
```

Next there are some notes about the syntax and semantics in the context of these examples. Since OCaml supports the functional and imperative paradigms, it distinguishes between immutable variables (actually not present in the programs shown) and memory references, therefore the different syntax for the binding of each one:

$$let\ x = 3\ (variable)$$

$$let\ y = ref\ 4\ (reference)$$

The *rec* keyword tells the compiler/interpreter that it is fine if the function calls itself, reducing the risk of the programmer using recursion by accident. OCaml is also sharp enough to understand these programs correctly even if we did not specify the types of the function or its arguments. Finally, the *let* keyword can be used to define a function or create a binding, something rather uncommon for someone with an imperative background.

2.3 Why3

Why3 is a platform whose objective is the verification of programs in a deductive way, presenting it self as a *"a front-end to third-party theorem provers"* [12]. These several theorem provers can be put to work together and vary greatly in nature, ranging from SMT solvers (for example, Z3 or cvc5) to TPTP provers and even interactive proof assistants (such as Coq or Isabelle). Under the hood, Why3 is an OCaml library, implemented in the form of an API. It features a CLI, a GUI and a benchmark tool to compare the performance of the different provers available.

2.4 GOSPEL

GOSPEL [13] (Generic Ocaml SPEcification Language) is a tool-agnostic specification language for OCaml that allows one to verify the correctness of a program (there are other purposes but this is the most interesting for this work). GOSPEL is based in Separation Logic and significantly improves the experience of the people that write or read the specifications, by making them much more concise. Let us reconsider the code presented in this subsection, but this time also show the GOSPEL specification for that program:

GOSPEL + OCaml

```
(*@ function rec gcd (a: int) (b:int) : int =
    if b = 0 then a
    else gcd b (mod a b) *)
(*@ requires a ≥ 0
    requires b ≥ 0
    variant b *)
```

GOSPEL + OCaml

```
let gcd_iter (a0: int) (b0: int) : int =
  let b = ref b0 in
  let a = ref a0 in
  while !b ≠ 0 do
      (*@ invariant 0 ≤ !b
          invariant 0 ≤ !a
          invariant gcd a0 b0 = gcd !a !b
          variant !b *)
      let tmp = !a in
      a := !b;
      b := tmp mod !b
  done;
  !a
(*@ result = gcd_iter a0 b0
    requires a0 ≥ 0
    requires b0 ≥ 0
    ensures result = gcd a0 b0*)
```

Although this is a simple program, we can already notice what a typical GOSPEL specification looks like. If we are dealing with a functional piece of code usually there will be a few *requires* clauses (could be none as well) at the beginning of the specification, representing the pre-conditions. After that, since these implementations use recursion most of the time, there will be one or more *variant* clauses, indicating what variables vary each the function is called. This serves as a guarantee that the recursion does not go forever. In this implementation, *variant b* is enough but we could eventually reason about how *a* varies over the execution of the program. Regarding the pre-conditions, this particular example states that *a* and *b* need to be non negative integer numbers, since the greatest common divisor of any two negative numbers will always be 0. Finally, we have a single post-condition *ensures result = gcd a0 b0* that guarantees that for any input, the two different implementations will output the same result.

2.5 Cameleer

Cameleer is a tool that aids automated deductive verification of OCaml programs. It relies on GOSPEL for the specification of the code and on Why3 to effectively verify the program. The limitations of the powerful Why3 tool and the reason for the creation of Cameleer start here: Why3 does not accept code that is not written in its intermediary language: WhyML. Now imagine wanting to verify your codebase with years and years of contributions, with maybe more than a million lines of code. Why3 would force you to translate OCaml into WhyML, *all by hand*. And only after that you would write the specification. Two obvious things come to mind: no one wants to do that job and if someone had to do it, they would very quickly start wondering if there is a tool that could help them, or even better, do that repetitive work instead of them. So, how is Cameleer able to save those poor programmers?

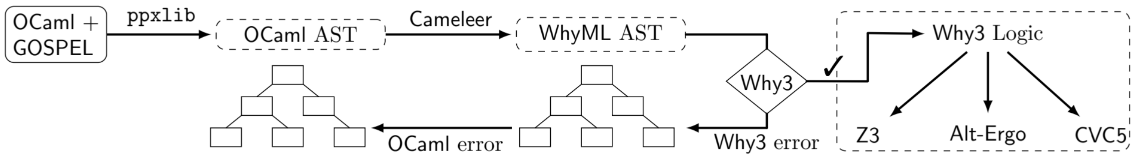

Figure 2.4: Architecture and verification pipeline, taken from [14].

The Cameleer pipeline is constituted by 4 different steps, the last one being different depending on whether Why3 detects errors in the translated code or not. The first step is to parse and change the abstract syntax tree of the input OCaml program (that contains GOSPEL annotations), using *ppxlib*, a library that is part of the GOSPEL ecosystem[3]. In the second step, the GOSPEL annotations are processed by a special parser/type-checker that converts the specification into nodes that become part of the previous OCaml AST. The third step consists in Cameleer itself translating the complete AST into a description in WhyML that is equivalent; Why3 then takes that as input. In step four, if the Why3 type-and-effect system is not able to validate the input generated in the previous step, the problems are shown in the context of the input OCaml code, the one that is submitted by the user. Alternatively, if no errors are found, a series of verification conditions are the output of the respective VCGEN. Those VCs should then be unraveled by the solvers featured in Cameleer or, if they are not able to do all the work automatically, this tool also allows human intervention to discharge more complex statements.

[3] https://github.com/ocaml-gospel/gospel

3

State of the Art

3.1 Self-composition

The proposal of self-composition in this [15] work was to offer an extensible and flexible way of controlling information flow. This is usually done by information flow type systems, but they suffer from the issues that self-composition attempted to solve. In the work mentioned above, the authors were trying to address the static enforcement of information flow policies but focused on a specific one called non-interference. Non-interference separates the state of a program into a public and a secret part and, by observing its execution, determines if there was a information leak on the secret part of the state.

Despite the well-defined focus, the efforts of the work mentioned in the beginning of the section apply to several programming languages and different definitions of security, even including declassification, partially. In the end, the authors were able to establish a general theory of self-composition to prove that programs are non-interfering. One of the main features of self-composition is its expressiveness and that there is no need to prove the soundness of type systems, since it is based on logic.

3.2 Cross-products

Cross-products [16] aim to prove program equivalence, with a strong focus on verifying compiler optimizations. Instead of proving that both programs are equivalent, the analysis is done by combining the two input programs into one: the cross-product. With them, instead of recurring to program analysis and proof rules developed specifically for translation validation, it became possible to use existing methods and tools to prove properties of a single program. Despite handling the most common intraprocedural compiler optimizations, cross-products can not be applied to two input programs with dissimilar structures, which is a major constraint for this work.

One important aspect of CoVaC, a framework developed by the authors of the aforementioned article, is that it was made to validate program equivalence in general while balancing precision and efficiency. The approach was to divide the analysis in two phases, a faster one first and a more precise after that. The first phase utilizes a fast, yet imprecise value numbering algorithm whose results are used to determine if it is necessary to apply the second phase. This final part is based on assertion checking, a static program verification method which, under the hood, computes the weakest-precondition [17] using CVC3 [18].

The results presented on the cited work reveal that CoVaC was able to verify a very considerable set of optimizations of LLVM [19], a complex modern compiler. Yet, the tool does not support verification of interprocedural optimizations or information flow policies, for example, but these are issues that the authors showed interest in addressing.

3.3 Product Programs

3.3.1 Motivation

Relational Hoare Logic serves as a good starting point to compare the behavior of two different executions of the same program or even two different programs. However, there are few available tools and practical reasoning logics for relational verification. One of the main limitations of the existing ones [4, 20] is the constraint of *structural equivalence*. Inversely, traditional program verification has diverse and extensive tool support. Therefore, as a way of getting around that obstacle, relational verification tasks could be soundly reduced into standard verification. This means we would translate Hoare quadruples ($\{\Phi\}$ *P1* ~ *P2* $\{\Psi\}$) into Hoare triples ($\{\phi\}$ *P* $\{\psi\}$), making the triples valid when the quadruples are also valid. Considering $\models$ the symbol for validity, the objective is finding ϕ, *P*, ψ that:

$$\models \{\phi\}\, \mathbf{P}\, \{\psi\} \quad \rightarrow \quad \models \{\Phi\}\, \mathbf{P1} \sim \mathbf{P2}\, \{\Psi\}$$

Let us consider two imperative and *separable* programs *P1* and *P2*. This enables the capacity of the assertions to be seen as first-order formulas about the variables of *P1* and *P2*. Self-composition [15] comes from the establishment of the wanted equalities above: $P \equiv P1;P2$, $\phi \equiv \Phi$ and $\psi \equiv \Psi$. Despite being sound and considerably complete, this construction is impractical [21] for two main reasons. One of them is that the existing automatic safety analysis tools available are not powerful enough to verify most of realistic problems and, when they can, there is a lack of performance. The other reason is that, for the authors of the last cited work, the safety analysis developed for naturally 1-safety problems is not expected to advance significantly in the foreseeable future.

There is another relevant method previously discussed in this section, the cross-products. These suffer from the constraint of structural equivalence, making impossible to reason about properties or program optimizations that are based on different control flows.

Product Programs [22] appear to be the best path to follow, since they represent a general notion that combines the flexibility of self-composition in executing asynchronous steps and the efficiency of cross-products when it comes to treat synchronous steps.

3.3.2 From relational verification to standard verification

Consider two generic programs *P1* and *P2* and their product program *P*. The approach of product programs to reduce relational reasoning into standard reasoning goes through the capacity of constructing *P*, that simulates the execution of both *P1* and *P2*. Next, we will describe the grammar rules to show how we will proceed and then explain how product programs are effectively constructed.

3.3.2.1 Establishing the ground rules

The commands in our programming model will stick to these grammar rules:

c ::= x:= e | a[e]:= e | **skip** | **assert**(b) | c; c | **if** b **then** c1 **else** c2 | **while** b **do** c

In this context, *x* is a integer variable, *a* is an array variable, *e* is an integer expression and *b* is a boolean expression. Execution states are represented as $S = (V_i \rightharpoonup \mathbb{Z})\ X\ (V_a \rightharpoonup (\mathbb{Z} \rightharpoonup \mathbb{Z}))$. The semantics of the commands are standard, deterministic and are based on the relation $\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle$. Notice that $\langle$**skip**, $\sigma\rangle$ marks the end of the programs, **assert**(b) blocks the execution of the program if *b* is false and we let $\langle c, \sigma\rangle \Downarrow \sigma'$ mean $\langle c, \sigma\rangle \rightsquigarrow^* \langle$**skip**, $\sigma'\rangle$.

As we discussed before in this document, two commands are *separable* if they operate on disjoint sets of variables. Similarly, we consider that two states are *separable* if their domains are disjoint. Consider $\mu_1 \uplus \mu_2$ represent the union of finite maps:

$$(\mu_1 \uplus \mu_2)\ x = \begin{cases} \mu_1 x & \text{if } x \in \text{dom}(\mu_1) \\ \mu_2 x & \text{if } x \in \text{dom}(\mu_2) \end{cases}$$

and overload this notation for the union of separable states $(\mu,\nu) \uplus (\mu',\nu')$, whose definition is $(\mu \uplus \mu', \nu \uplus \nu')$. Taking into account that we assume that the states are separable, another way of looking at assertions is by viewing them as relations on states: $(\sigma_1, \sigma_2) \in [\![\phi]\!]$ iff $\sigma_1 \uplus \sigma_2 \in [\![\phi]\!]$. Therefore, the definition below illustrates the formal statement of valid relational specifications.

Definition 1 - *Two commands c_1 and c_2 satisfy the pre-condition ϕ and the post-condition ψ described by a valid Hoare quadruple if, for all states σ_1, σ_2, σ_1', σ_2' such that $\sigma_1 \uplus \sigma_2 \in [\![\phi]\!]$ and $\langle c_1, \sigma_1\rangle \Downarrow \sigma_1'$ and $\langle c_2, \sigma_2\rangle \Downarrow \sigma_2'$, we have $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.*

Since our objective is to reduce the validity of Hoare quadruples to validity of Hoare triples, we must say that we establish our notion of valid Hoare triple as stronger than usual. It requires that the command allows the program to finish its execution, i.e., the command is *non-stuck*.

Definition 2 - *A Hoare triple ϕ c ψ is valid ($\models \{\phi\}$ c $\{\psi\}$) if c is ϕ-nonstuck and for all σ, σ' $\in S$, $\sigma \in [\![\phi]\!]$ and $\langle c, \sigma\rangle \Downarrow \sigma'$ imply $\sigma' \in [\![\psi]\!]$.*

This notion of validity requires, however, that we extend Hoare logic for it to be able to treat **assert** statements. The necessary rule is:

$$\frac{}{\vdash\ \{b \wedge \Phi\}\ \textbf{assert}(\mathrm{b})\ \{\Phi\}}$$

3.3.2.2 Construction of Product Programs

Firstly in this section, we define the rules that will be used to deal with structurally equivalent programs. After that, we that set with structural transformations to allow the treatment of programs with different structures.

$$\frac{}{c_1 \times c_2\ \rightarrow\ c_1; c_2} \qquad \frac{c_1 \times c_2\ \rightarrow\ c \qquad c_1' \times c_2'\ \rightarrow\ c'}{(c_1; c_1') \times (c_2; c_2')\ \rightarrow\ c; c'}$$

$$\frac{c_1 \times c_2\ \rightarrow\ c}{(\textbf{while}\ b_1\ \textbf{do}\ c_1) \times (\textbf{while}\ b_2\ \textbf{do}\ c_2)\ \rightarrow\ \textbf{assert}(b_1 \Leftrightarrow b_2);\ \textbf{while}\ b_1\ \textbf{do}\ (c; \textbf{assert}(b_1 \Leftrightarrow b_2))}$$

$$\frac{c_1 \times c_2\ \rightarrow\ c \qquad c_1' \times c_2'\ \rightarrow\ c'}{(\textbf{if}\ b_1\ \textbf{then}\ c_1\ \textbf{else}\ c_1') \times (\textbf{if}\ b_2\ \textbf{then}\ c_2\ \textbf{else}\ c_2')\ \rightarrow\ \textbf{assert}(b_1 \Leftrightarrow b_2);\ \textbf{if}\ b_1\ \textbf{then}\ c\ \textbf{else}\ c'}$$

$$\frac{c_1 \times c\ \rightarrow\ c_1' \qquad c_2 \times c\ \rightarrow\ c_2'}{(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \times c\ \rightarrow\ \textbf{if}\ b\ \textbf{then}\ c_1'\ \textbf{else}\ c_2'}$$

Figure 3.1: Product construction rules.

The figure describes a set of rules to derive a product construction judgment of the type $c_1 \times c_2 \rightarrow c$. To make sure that c simulates with precision the behavior of c_1 *and* c_2, the construction of products introduces **assert** statements. During the phase of the verification of the program, these validation constraints are viewed as local assertions and discharged. Let us consider the rule that synchronizes two loops, for example: **assert**($\mathrm{b}_1 \Leftrightarrow \mathrm{b}_2$) is necessary to guarantee that the number of iterations of each loop is the same. This achieves what we aimed for, that is, the product program can be verified with standard logic.

Proposition 1 - *For all statements c_1 and c_2, pre-condition ϕ and post-condition ψ, if $c_1 \times c_2 \rightarrow c$ and $\models \phi\ c\ \psi$ then $\models \phi\ c_1 \sim c_2\ \psi$.*

In other words, if c is the resulting product program of *c1* and *c2*, then we can reason about the validity of the relational judgment between *c1* and *c2* through the validity of the standard judgment of c.

As we mentioned before, the rules present in the previous figure are limited to structurally equivalent programs. For example, if we consider two programs each with a loop and their guards are not equivalent, the product would have to be sequentially composed. The structural translations proposed extend the construction of the already defined products in the form of a refinement relation, with a judgment of the form $c \succcurlyeq c'$. This means that every execution of c is an execution of c' except when the latter's execution does not terminate:

Definition 3 - *A command c' is a refinement of c, if, for all states σ, σ':*

- *1. if $\langle c', \sigma\rangle \Downarrow \sigma'$ then $\langle c, \sigma\rangle \Downarrow \sigma'$*
- *2. if $\langle c, \sigma\rangle \Downarrow \sigma'$ then either the execution of c' with initial state σ gets stuck, or $\langle c', \sigma\rangle \Downarrow \sigma'$.*

In the figure is described the set of rules that define judgments of the form $\vdash c \succcurlyeq c'$.

$$\frac{}{\vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \succcurlyeq \textbf{assert}(b);\ c_1} \qquad \frac{}{\vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \succcurlyeq \textbf{assert}(\neg b);\ c_2}$$

$$\frac{}{\vdash \textbf{while } b \textbf{ do } c \succcurlyeq \textbf{assert}(b);\ c;\ \textbf{while } b \textbf{ do } c} \qquad \frac{}{\vdash \textbf{while } b \textbf{ do } c \succcurlyeq \textbf{while } b \wedge b' \textbf{ do } c}$$

$$\frac{}{\vdash \textbf{while } b \textbf{ do } c \succcurlyeq \textbf{assert}(b);\ c;\ \textbf{assert}(\neg b)} \qquad \frac{\vdash c \succcurlyeq c'}{\vdash \textbf{while } b \textbf{ do } c \succcurlyeq \vdash \textbf{while } b \textbf{ do } c'}$$

$$\frac{\vdash c_1 \succcurlyeq c_1' \qquad \vdash c_2 \succcurlyeq c_2'}{\vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \succcurlyeq \textbf{if } b \textbf{ then } c_1' \textbf{ else } c_2'} \qquad \frac{\vdash c \succcurlyeq c' \qquad \vdash c' \succcurlyeq c''}{\vdash c \succcurlyeq c''} \qquad \frac{}{\vdash c \succcurlyeq c}$$

$$\frac{\vdash c_1 \succcurlyeq c_1' \qquad \vdash c_2 \succcurlyeq c_2'}{\vdash c_1; c_2 \succcurlyeq c_1'; c_2'}$$

Figure 3.2: Syntactic reduction rules.

One can see that the rules point that the executions of c and c' match for every initial state that makes the introduced **assert** statements valid. It is possible to prove that the judgment $c \succcurlyeq c'$ maintains a refinement relation by showing that, for every assertion ϕ, if c′ is ϕ-nonstuck, then for all $\sigma \in [\![\phi]\!]$ such that $\langle c, \sigma\rangle \Downarrow \sigma'$ we have $\langle c', \sigma\rangle \Downarrow \sigma'$. We add one more rule that makes a preliminary refinement transformation over the product components:

$$\frac{\vdash c_1 \succcurlyeq c_1' \qquad \vdash c_2 \succcurlyeq c_2' \qquad c_1' \times c_2' \;\rightarrow\; c}{c_1 \times c_2 \;\rightarrow\; c}$$

This does not invalidate **Proposition 1**. Finally, we formally reduce the problem of proving the validity of a relational judgment into the construction of the product program followed by the standard verification of that same product program.

Proposition 2 - *For all statements c_1 and c_2, pre-condition ϕ and post-condition ψ, if $c_1 \times c_2$ $\rightarrow c$ and $\vdash \phi\ c\ \psi$ then $\models \phi\ c_1 \sim c_2\ \psi$.*

3.3.3 Examples

Although product programs can be applied to several relational properties, such as non-interference and continuity, we will focus our examples on the correctness of program optimizations.

3.3.3.1 Loop alignment

Loop alignment is an optimization that consists in improving cache effectiveness by increasing the proximity of the memory locations accessed in each iteration of the loop.

Consider $N \geq 1$. In the source program, note that, in each iteration, the array b is accessed twice; firstly in the write to index i and secondly in the read of position *i-1*. What loop alignment does is transform the access of different indexes of b into only one index, in this case, i. The corresponding product program starts by ensuring that the number of iterations of the loop is the same for the source and optimized programs, through the assignment statement $\overline{d}[1] := \overline{b}[0]$. To be verified, the program product needs a pre-condition, a post-condition and a loop invariant. The pre-condition is $a = \overline{a} \wedge b[0] := \overline{b}[0]$. The post-condition is $d[1, N] = \overline{d}[1, N]$, meaning that, for the indexes in the interval $[1, N]$, the values of the arrays d and $\overline{d}$ are the same. A suitable loop invariant is $d[1, i) = \overline{d}[1, i) \wedge b[j] = a[j] \wedge \overline{b}[i] = b[i] \wedge i = j + 1$. This specification guarantees that, if the input arrays a and b are equal, the values present in the array d are the same when the execution of both product components terminates.

Source program

```
i := 1;
while (i ≤ N) do
  b[i] := a[i];
  d[i] := b[i-1];
  i++
```

Optimized program

```
j := 1;
d̄[1] := b̄[0];
while (j ≤ N-1) do
  b̄[j] := ā[j];
  d̄[j+1] := b̄[j];
  j++;
b̄[N] := ā[N]
```

Product program

```
{a = ā ∧ b[0] = b̄[0]}
  i := 1;
  j := 1;
  assert(i ≤ N);
  b[i] := a[i]; d[i] := b[i-1]; i++;
  d̄[1] := b̄[0];
  assert(i ≤ N ⇔ j ≤ N-1);
  while (i ≤ N) do
    b[i] := a[i]; d[i] := b[i-1]; i++;
    b̄[j] := ā[j]; d̄[j+1] := b̄[j]; j++;
    assert(i ≤ N ⇔ j ≤ N-1);
  b̄[N] := ā[N]
{d[1,N] = d̄[1,N]}
```

Figure 3.3: Loop alignment.

3.3.3.2 Induction variable strength reduction

Product programs allow the verification of optimizations that *maintain* the control flow of programs, but simply modify the basic blocks. Some rather common examples of this are constant propagation and common subexpression elimination.

The figure demonstrates the effects that applying strength reduction has in a small program. In the example, j is a derived induction variable defined as a linear function on the induction variable i. The optimization in question substitutes the statement $j := i * B + C$ by the equivalent and more performant statement $j' \mathrel{+}= B$ (multiplication is more costly than addition to a computer), makes the assignment $x' \mathrel{+}= j'$ come first inside the loop and adds an initial assignment $j' = C$ before the start of the loop. To verify the correctness of the optimization, we need to guarantee that $x = x'$ is an invariant of the product program. And to do that, we need the linear condition $j := i * B + C$ to be part of the loop invariant.

Source program

```
i := 0;
while (i < N) do
  j := i * B + C;
  x += j;
  i++
```

Optimized program

```
i´ := 0;
j´ := C;
while (i´ < N) do
  x´ += j´;
  j´ += B;
  i´++
```

Product program

```
i := 0; i´ := 0; j´ := C;
while (i < N ∧ i´ < N) do
  j := i * B + C; x += j; i++
  x´ += j´; j´ += B; i´++
```

Figure 3.4: Induction variable strength reduction.

3.4 Program Equivalence

Program equivalence stands out as a crucial subject within the field of formal verification [23]. "Traditional" program correctness by itself is usually harder to prove than program equivalence, especially when it comes to real world programs, making the latter a topic whose research can reach success easier.

This work [24] brings to light an interesting approach called regression verification, which combines regression testing with formal verification. It is based on invariant inference techniques to automatically prove that two imperative pointer programs are equivalent, in the context of one being an updated version of the other. By updated version we mean that the program has now more features, has been refactored or went through performance optimizations.

This automatic proof method is achieved by first transforming the two programs into Horn clauses over uninterpreted predicate symbols. Then, these clauses constrain equivalence witnessing coupling predicates that establish the relation of the two programs at key points. Finally, if there is a solution for the coupling predicates, a Horn constraint solver is utilized to find it. It may be important to mention that, according to the authors, the approach was implemented and its effectiveness demonstrated.

Besides the fact that the focus of our work does not include pointer programs, it is important to note that, due to the lack of human-written specification, this approach has its limitations. The range of situations where regression verification can be applied is still confined to some programming language constructs and there are issues regarding the scalability.

4 Preliminary Results

Proving the equivalence of two different OCaml programs can be a hard task, depending on the complexity and size of said programs. Therefore, we implemented a small set of simple programs that show two different ways of establishing program equivalence. The first one relies on including a post-condition that states that the result of program P_1 is equal to the result of program P_2 for the same input. The second way uses product programs, where we present the source and the transformed programs. Then, we apply these rules to create the corresponding product program, which can be proven with standard verification.

4.1 Simple equivalence proofs

In this section, we present the functional and imperative implementations of each algorithm. We also compare the specification of each implementation, pointing the similarities and differences. All of the following programs have been verified using Cameleer and the information in the tables was provided by Why3.

4.1.1 Factorial

Both functional and imperative implementations of the factorial have only one pre condition *n >= 0*. They differ, however, in the post condition, which is *result = fact n* in the case of the imperative implementation and non-existent in the case of the functional implementation. Moreover, the functional implementation has a variant *n*, while the imperative implementation has an invariant, *!res = fact (i-1)*.

GOSPEL + OCaml

```
(*@ function rec fact (n: integer) : integer =
if n = 0 then 1 else n * fact (n-1) *)
(*@ requires n ≥ 0
  variant n *)
```

OCaml

```
let fact_iter (n: int) : int =
  if n ≤ 1 then 1
  else
    begin
      let res = ref 1 in
      for i = 2 to n do
        (*@ invariant !res = fact (i-1) *)
        res := !res * i
      done;
      !res
    end
(*@ result = fact_iter n
  requires n ≥ 0
  ensures result = fact n *)
```

Regarding the proof of both the implementations of the factorial, Z3 was able to discard all of the verification conditions (VCs) automatically. Furthermore, this SMT solver took a quarter of a second to complete the whole proof, verifying the functional implementation in 0.08 seconds and the imperative implementation in 0.17 seconds.

Proof obligations		**Z3 4.13.0**
lemma `VC for fact`	lemma `variant decrease`	0.03
	lemma `precondition`	0.05
lemma `VC for fact_iter`	lemma `postcondition`	0.04
	lemma `loop invariant init`	0.05
	lemma `loop invariant preservation`	0.04
	lemma `postcondition`	0.01
	lemma `VC for fact_iter`	0.03

Table 4.1: Factorial implementations verification results.

4.1.2 Fibonacci

Both functional and imperative implementations of the factorial have only one pre condition *n >= 0*. They differ, however, in the post condition, which is *result = fib n* in the case of the imperative implementation and non-existent in the case of the functional implementation. Moreover, the functional implementation has a variant *n*, while the imperative implementation has two invariants, *!prev = fib (i-2)* and *!res = fib (i-1)*.

GOSPEL + OCaml

```
(*@ function rec fib (n: integer) : integer =
if n ≤ 1 then n else fib (n-1) + fib (n-2) *)
(*@ requires n ≥ 0
  variant n *)
```

OCaml

```
let fib_iter (n: int) : int =
  if n ≤ 1 then n
  else
    begin
      let prev = ref 0 in
      let res = ref 1 in
      let temp = ref 1 in

      for i = 2 to n do
        (*@ invariant !prev = fib (i-2)
          invariant !res = fib (i-1) *)
        temp := !res;
        res := !res + !prev;
        prev := !temp;
      done;

      !res
    end
(*@ result = fib_iter n
  requires n ≥ 0
  ensures result = fib n *)
```

Using Z3 as we did with the factorial, we were able to prove the correctness of both implementations without any human intervention, taking a total of 0.3 seconds. Besides that, the SMT solver spent 0.13 seconds verifying the functional implementation and 0.17 seconds verifying the imperative implementation.

4.2 Equivalence proofs using product programs

In this section, we present a program and a slightly different version. Then we combine them using the technique of product programs, more specifically, these rules. Finally, we verified those product programs using the Why3 tool. The results present on the tables were generated by that tool. All the code is written in WhyML.

Proof obligations		Z3 4.13.0
lemma `VC for fib`	lemma `variant decrease`	0.03
	lemma `precondition`	0.03
	lemma `variant decrease`	0.03
	lemma `precondition`	0.04
lemma `VC for fib_iter`	lemma `postcondition`	0.03
	lemma `loop invariant init`	0.02
	lemma `loop invariant init`	0.03
	lemma `loop invariant preservation`	0.02
	lemma `loop invariant preservation`	0.03
	lemma `postcondition`	0.01
	lemma `VC for fib_iter`	0.03

Table 4.2: Fibonacci implementations verification results.

4.2.1 Program based on assignments

The (original) program is based on only two assignment instructions: *y := x + 1;* and *z := y + 1;*. Despite being a very simple example, it is enough to show the usage of the product construction rules in practice.

To make the programs separable, we renamed *x*, *y* and *z* in the original program to *x1*, *y1* and *z1* and in the transformed program to *x2*, *y2* and *z2*. The product program establishes that, if the values of the variables *x1* and *x2* are the same before the function *product* starts, it is guaranteed that *y1 = y2* and *z1 = z2*. And, since there is no assignment to any of the x-components, *x1 = x2* will also be true if the pre-condition is respected.

There was no need for human intervention in verifying the product program, as the SMT solver Z3 completed the proof in under a second.

Proof obligations		Z3 4.13.0
lemma `VC for product`	lemma `postcondition`	0.01
	lemma `postcondition`	0.00

Table 4.3: Program based on assignments verification results.

Original program

```
use int.Int                                WhyML

val ref x : int
val ref y : int
val ref z : int

let original ()
  =
  y ← x + 1;
  z ← y + 1;
```

Transformed program

```
use int.Int                                WhyML

val ref x : int
val ref y : int
val ref z : int

let transformed ()
  =
  z ← x + 2;
  y ← z - 1;
```

Product program

```
use int.Int                                                      WhyML

val ref x1 : int
val ref x2 : int
val ref y1 : int
val ref y2 : int
val ref z1 : int
val ref z2 : int

let product ()
  requires { x1 = x2 }
  ensures { y1 = y2 }
  ensures { z1 = z2 }
  =
  y1 ← x1 + 1;
  z2 ← x2 + 2;
  z1 ← y1 + 1;
  y2 ← z2 - 1;
```

Figure 4.1: Program based on assignments.

4.2.2 Program with a while loop

This program, already mentioned before, is slightly more complex than the example before, as this one features a while loop. The optimization here comes from the fact that, for a computer, a multiplication is more expensive than an addition, especially when performed inside a loop. This is called induction variable strength reduction.

The product program does not need a pre-condition, but its post.condition states that $x = x'$ at the end of the execution. So, how is this achieved? Well, with one variant to prove termination and three invariants to guarantee the post-condition; one of them is the exact same comparison (*invariant* $\{x = x'\}$).

The proof was completed automatically by Z3 and the total duration was 0.16 seconds.

Original program

WhyML

```
use int.Int

let source (b c n: int) : (x: int)
= let ref i = 0 in
  let ref j = 0 in
  let ref x = 0 in
  while i < n do
    j ← i * b + c;
    x ← x + j;
    i ← i + 1;
  done;
  x
```

Transformed program

WhyML

```
use int.Int

let transformed (b c n: int) : (x': int)
= let ref i' = 0 in
  let ref j' = c in
  let ref x' = 0 in
  while i' < n do
    x' ← x' + j';
    j' ← j' + b;
    i' ← i' + 1
  done;
  x'
```

Product program

WhyML

```
use int.Int

let product (b c n: int) : (x: int, x': int)
  ensures { x = x' }
= let ref i = 0 in
  let ref i' = 0 in
  let ref j = 0 in
  let ref j' = c in
  let ref x = 0 in
  let ref x' = 0 in
  while i < n && i' < n do
    variant   { n - i }
    invariant { j' = i' * b + c }
    invariant { i = i' }
    invariant { x = x' }
    j ← i * b + c;
    x ← x + j;
    i ← i + 1;
    x' ← x' + j';
    j' ← j' + b;
    i' ← i' + 1
  done;
  x, x'
```

Figure 4.2: Program with a while loop.

	Proof obligations	**Z3 4.13.0**
lemma `VC for main`	lemma `loop invariant init`	0.01
	lemma `loop invariant init`	0.02
	lemma `loop invariant init`	0.02
	lemma `loop variant decrease`	0.02
	lemma `loop invariant preservation`	0.03
	lemma `loop invariant preservation`	0.01
	lemma `loop invariant preservation`	0.04
	lemma `postcondition`	0.01

Table 4.4: Program with a while loop verification results.

5 Work plan

- *Task 1: Collection of case studies* In March, we plan to select interesting cases where applying our work would be beneficial. Some examples are compiler optimizations (and optimizations in general), reduction of higher-order functions to first-order and even code versioning, in the sense that refactoring or new features do not modify the behavior of previously correct code.
- *Task 2: First-order product programs for OCaml* Afterwards, and spanning across a month and a half, we will adapt the original work of product programs (which is based on the *while* language) to the reality of OCaml, in the first-order context.
- *Task 3: First-order implementation in Cameleer* During two months, we plan on implementing the first-order OCaml approach in the Cameleer tool.
- *Task 4: Extension of product programs to higher-order* After that, in the month of July, we plan to extend our notion of product programs in OCaml to include some higher-order constructs focused on iteration, such as folds (left and right), map and iter.
- *Task 5: Higher-order implementation in Cameleer* In the following month, we will attempt to extend the implementation of product programs in Cameleer to include the higher-order functions mentioned in the previous task.
- *Task 6: Dissertation Writing* Finally, we will dedicate the entire month of September to write the dissertation.

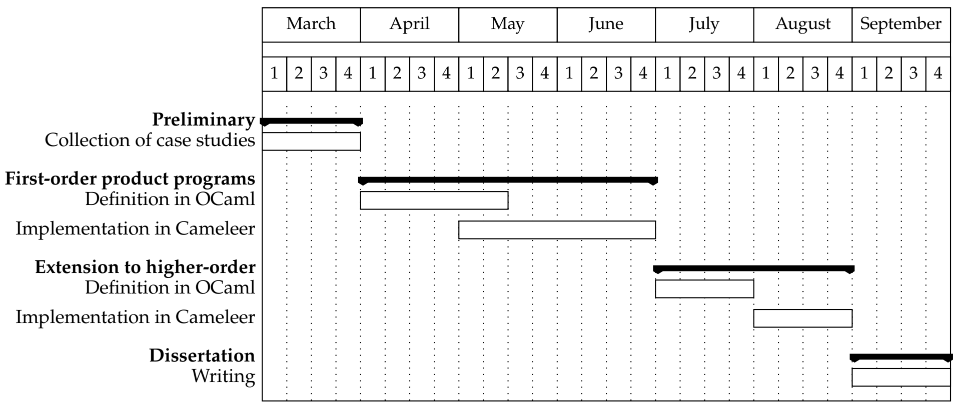

Figure 5.1: Planned Schedule.

Bibliography

[1] M. Brain and E. Polgreen. "A Pyramid Of (Formal) Software Verification". In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*. Ed. by A. Platzer et al. Vol. 14934. Lecture Notes in Computer Science. Springer, 2024, pp. 393–419. doi: 10.1007/978-3-031-71177-0_24. url: https://doi.org/10.1007/978-3-031-71177-0%5C_24 (cit. on pp. 1, 3).

[2] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969-10), pp. 576–580. issn: 0001-0782. doi: 10.1145/363235.363259. url: https://doi.org/10.1145/363235.363259 (cit. on p. 3).

[3] D. A. Naumann. *Thirty-seven years of relational Hoare logic: remarks on its principles and history*. 2022. arXiv: 2007.06421 [cs.LO]. url: https://arxiv.org/abs/2007.06421 (cit. on p. 4).

[4] N. Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by N. D. Jones and X. Leroy. ACM, 2004, pp. 14–25. doi: 10.1145/964001.964003. url: https://doi.org/10.1145/964001.964003 (cit. on pp. 4, 13).

[5] *OCaml's Reference Manual*. https://ocaml.org/manual/5.3/index.html. Accessed on February 6, 2025 (cit. on p. 7).

[6] Oracle. *Java Programming Language Enhancements in Java SE 8*. Accessed on February 6, 2025. n.d. url: https://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8 (cit. on p. 8).

[7] JetBrains. *Kotlin functions documentation*. Accessed on February 6, 2025. Last Modified on September 25, 2024. url: https://kotlinlang.org/docs/functions.html (cit. on p. 8).

[8] Anonymous. *Who Uses OCaml?* Accessed on February 6, 2025. n.d. url: https://ocaml.org/industrial-users (cit. on p. 8).

[9] Anonymous. *An SMT Solver for Software Verification*. Accessed on February 6, 2025. n.d. url: https://alt-ergo.ocamlpro.com/ (cit. on p. 8).

[10] Anonymous. *What is Coq?* Accessed on February 6, 2025. n.d. URL: https://coq.inria.fr/about-coq (cit. on p. 8).

[11] Reason. *Messenger.com Now 50% Converted to Reason*. Accessed on February 6, 2025. September 8, 2017. URL: https://reasonml.github.io/blog/2017/09/08/messenger-50-reason (cit. on p. 8).

[12] F. Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland, 2011, pp. 53–64. URL: https://inria.hal.science/hal-00790310 (cit. on p. 9).

[13] 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).

[14] 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. 11).

[15] G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure Information Flow by Self-Composition". In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.17. URL: https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.17 (cit. on pp. 12, 13).

[16] A. Zaks and A. Pnueli. "CoVaC: Compiler Validation by Program Analysis of the Cross-Product". In: *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*. Ed. by J. Cuéllar, T. S. E. Maibaum, and K. Sere. Vol. 5014. Lecture Notes in Computer Science. Springer, 2008, pp. 35–51. DOI: 10.1007/978-3-540-68237-0_5. URL: https://doi.org/10.1007/978-3-540-68237-0%5C_5 (cit. on p. 12).

[17] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://www.worldcat.org/oclc/01958445 (cit. on p. 13).

[18] C. Barrett and C. Tinelli. *CVC3 is an automatic theorem prover for Satisfiability Modulo Theories (SMT) problems*. Accessed on February 9, 2025. n.d. URL: https://cs.nyu.edu/acsys/cvc3/ (cit. on p. 13).

[19] G. Barthe, P. R. D'Argenio, and T. Rezk. *The LLVM Project is a collection of modular and reusable compiler and toolchain technologies*. Accessed on February 9, 2025. n.d. URL: https://llvm.org/ (cit. on p. 13).

[20] H. Yang. "Relational separation logic". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 308–334. DOI: 10.1016/J.TCS.2006.12.036. URL: https://doi.org/10.1016/j.tcs.2006.12.036 (cit. on p. 13).

[21] T. Terauchi and A. Aiken. "Secure Information Flow as a Safety Problem". In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Ed. by C. Hankin and I. Siveroni. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367. DOI: 10.1007/11547662_24. URL: https://doi.org/10.1007/11547662%5C_24 (cit. on p. 13).

[22] G. Barthe, J. M. Crespo, and C. Kunz. "Relational Verification Using Product Programs". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by M. J. Butler and W. Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17. URL: https://doi.org/10.1007/978-3-642-21437-0%5C_17 (cit. on p. 14).

[23] O. Strichman. "Special issue: program equivalence". In: *Formal Methods Syst. Des.* 52.3 (2018), pp. 227–228. DOI: 10.1007/S10703-018-0318-Y. URL: https://doi.org/10.1007/s10703-018-0318-y (cit. on p. 19).

[24] V. Klebanov, P. Rümmer, and M. Ulbrich. "Automating regression verification of pointer programs by predicate abstraction". In: *Formal Methods Syst. Des.* 52.3 (2018), pp. 229–259. DOI: 10.1007/S10703-017-0293-8. URL: https://doi.org/10.1007/s10703-017-0293-8 (cit. on p. 19).