NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO FRANCISCO SERRENHO NINI

BSc in Computer Science

FORMAL VERIFICATION OF PROGRAMS EQUIVALENCE

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

February, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

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

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

Formal Verification of Programs Equivalence

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.3.1) [1].

Dedico esta tese (e todo o meu percurso académico) aos meus pais, Maria João e Paulo, e à minha namorada, Carolina. Por todo o apoio e por nunca terem duvidado de mim nem das minhas capacidades. Nem mesmo quando encarei os momentos mais sombrios, tenham sido eles causados pela exigência objetiva do curso, pressão dos que duvidavam de mim ou por obstáculos impostos por mim próprio, consciente ou inconscientemente.
A eles e a todos os que me apoiaram, de uma maneira ou doutra, por mais pequeno que tenha sido o gesto, o meu sincero **obrigado** - serão certamente lembrados.

Acknowledgements

I start by thanking Mário, my supervisor, for all of the time and work put into this thesis so that we could create something new, useful, and (in my biased perspective) wonderful. At the beginning, I doubted whether I could understand all of those complex papers and make a contribution to science, as small as the impact would be. However, confidence quickly built up when I put *as mãos na massa* and I had a much better time than I ever imagined I could developing a master's thesis. And, because of that, I blame Mário for having chosen a theme that fit me so well :)

I am also grateful for all the help that Ion gave me during this last year. He and all of my lab colleagues were definitely a motivating force that drove me to want to do things as incredible as they did. Furthermore, I would also like to thank professor Simão Melo de Sousa for being the rapporteur for this thesis.

Finally, I want to thank Faculdade de Ciências e Tecnologia and Universidade Nova de Lisboa for making all of this possible and for having an incredible culture, colleagues, and professors that taught me so much in such a welcoming way. And also for being located in a place that is so amazingly well-balanced for my taste that I proudly consider it my second-favorite place to live in the world, only behind the place where I was born and raised.

”

“All animals are equal, but some animals are more equal than others.”

— George Orwell

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 describe the details of our implementation and showcase the capabilities of our tool using several real world examples.

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, descreveremos os detalhes da nossa implementação e demonstraremos as capacidades da nossa ferramenta através de diversos exemplos realistas.

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 [2]. Additionally, there are other situations as dangerous as a plane crash: medical equipment failure [3], space accidents [4], defense systems errors [5]... 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 certain program outputs what is intended given its inputs, assuming the execution of that program finishes. 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. One of the most promising paths is **program equivalence**, a crucial subject within the field of formal verification [6].

1.2 Problem Definition

The question that this work focus its attention on is:

Can we automatically prove that two different programs are 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 [7] 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. So, when we mean automatic proof of equivalence, we mean no human interaction after writing the specification, although automatic inference of loop invariants and variants would definitely make the programmer's life much simpler.

In the context of human-assisted verification, and considering two versions of the same program, we first need to prove that the simpler program (in terms of specification) is correct, using already existent methods. Then, to prove that the more complex one is also correct, we establish a connection between the two. This approach may enable an easier and faster proof that programs are correct, constrating with the more common way of proving that the two are correct independently of each other. WhyRel [8] is an auto-active tool that aims to verify relational properties of pointer programs through relational region logic.

1.3 Goals and Contributions

This work aims to bring to OCaml [9] the concept of *biprograms*, facilitating the task of proving that two different programs are equivalent. Those programs are not expected to have identical structures, but since there is the need to reason about them jointly, there are still some restrictions explained later. To achieve this, our tool will steer towards the direction of the translation rules defined in WhyRel. The work on product programs [10] presents a good set of rules, but WhyRel's are more refined and easier to apply in practice. It is also important to mention that, in order to verify the generated program, we will use GOSPEL [11] to write the specification and Cameleer [12], so that the user can utilize Why3 [13] to finish the proof.

Since our tool is a transpiler and we aim to output OCaml code, we still need to clarify what will the input be like. The ideal source language would be one whose syntax is as close as possible to OCaml's, so that programmers used to the target language do not have a steep learning curve if they need to prove equivalence of their programs. WhyRel's biprograms language is a good starting point, since it guides us on what a language capable of reasoning about relational properties needs to feature. However, it has many differences to OCaml in relation to syntax. Therefore, we propose to define a new language that has enough features so that one can write functional code as done in OCaml, while disposing of mechanisms to reason about two programs in tandem.

Besides program equivalence, we explore if our tool can reason about other relational properties, such as program inequivalence or equivalence between corresponding memory segments (after the execution of the programs finish). To conduct a more in-depth examination, we also observe how different it is, in practice, to prove different relational properties. i.e., what does the user need to change in the specification or code in order to prove equivalence versus inequivalence for example.

1.4 Thesis Structure

- Chapter 2 presents the foundational concetps and tools underlying the scientific subject of this work.
- In chapter 3, we explain the most relevant developments regarding the subject of program equivalence proof: self-composition, cross-products, product programs and WhyRel. The most relevant ones for this thesis are the last two.
- Chapter 4 starts by describing the way we based our tool on WhyRel's translation rules. After that, we present the architecture of bip2ml and how it is integrated in the Cameleer pipeline. Next, we describe our biprograms language, BipLang, and discuss some implementations details. Finally, we demonstrate the application of the translation rules by our transpiler.
- In chapter 5, we showcase several examples of what our transpiler is capable of. We start with a simple program, its translation and proof. Then, we do the same but for real world examples that demonstrate the usefulness of bip2ml.
- Finally, chapter 6 summarizes our contributions, the limitations of our tool and possible future work.

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 [7] 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 [14] 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 we execute S in a state that satisfies a given pre-condition P, then the execution finishes in a state that 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:

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

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 [15, 16] (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 non-interference and the equivalence of two implementations of the same algorithm, the latter being the one we are most interested in.

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 P_1 and P_2 start in a state in which both respect the pre-condition (let us name it Φ), their executions will end (if they terminate) in a state in which both respect 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. That relation between the two programs is defined by the $\sim$ symbol. This idea is known as the Hoare quadruple:

$$\{\Phi\}\, P_1 \sim P_2\, \{\Psi\}$$

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 versions or the ones that are easier to prove. However, this thesis is definitely not the only one that relies on the connection between two different implementations of the same idea.

One common example are compiler optimizations, since they aim to make the code more performant but still outputting the same results. Another example is represented by a "client" program that uses a abstract data type and does not care about which implementation the "server" provides. This can happen since the implementations are expected to be observationally equivalent (i.e., to the "client" they behave the same way) and is the "server" internal logic that decides which one to choose. That choice would depend on some characteristics that, for example, cause one implementation to be more performant than some other(s) in a given situation.

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

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 $P_i.j$ that will be used in this subsubsection, where i represents a given program index, P_i refers to a given program and $P_i.j$ the identifier of a variable j in program P_i.

The first example corresponds to two different programs, let us call them P_1 and P_2, and they both contain only two assignment statements applied to the same group of variables (a, b, c):

P_1	P_2
$b := a + 1;$	$c := a + 2;$
$c := b + 1;$	$b := c - 1;$

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 P_1 and P_2, we need the pre-condition to assert that $P_1.a = P_2.a$:

$$\{\Phi\}\, P_1 \sim P_2\, \{\Psi\}$$

$$where \;\; \Phi \triangleq P_1.\mathrm{a} = P_2.\mathrm{a}$$
$$and \;\; \Psi \triangleq P_1.\mathrm{a} = P_2.\mathrm{a} \wedge P_1.\mathrm{b} = P_2.\mathrm{b} \wedge P_1.\mathrm{c} = P_2.\mathrm{c}$$

This Hoare quadruple should be read as: *For any executions of P_1 and P_2, if they guarantee the equality of their a-components, they both diverge from the post-condition or they both finish in states where their a-components, b-components and c-components have the same values* [2].

We now introduce the concept of separability. If two programs P_1 and P_2 utilize disjoint sets of variables, we say that they are *separable*. In this context, this means that these two programs have their own memory, so that the variables of P_1 are never accessed by P_2 and vice-versa. 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, P_1, whose variables are {a, b, i, j} and P_2 whose variables are {a', b', i', j'}, meaning that they are separable:

P_1	P_2
$a := 1;$	$a' := 1;$
$while\ a < b\ do$	$i' := j' * j';$
$\quad i := j * j;$	$while\ a' < b'\ do$
$\quad a := a * i;$	$\quad a' := a' * i';$
od	od

[2] The values of the components of any of the programs can differ between themselves ($P_1.a$ has no relation to $P_1.b$, for example).

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 P_2. 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:

$$\{\Phi\}\ P_1 \sim P_2\ \{\Psi\}$$

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

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 P_1 and P_2 hold the same values after the executions terminate, we must assert that the b-components and j-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 P_1 and P_2 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 P_1 and P_2. The way one should look at the transformation of Hoare triples into Hoare quadruples is that if $\{\Phi\}\ P_1 \sim P_2\ \{\Psi\}$ is true, then $\{\phi\}\ P_1\ \{\psi\}$ will be true as well.

2.1.2.3 Axioms and Inference Rules

Consider the following grammar, that will be used in the rules in this subsubsection. Let n represent any integer number {... -1, 0, 1 ...}, x any integer variable of a given program and a any array variable. Let e be the set of integer expressions, b the set of boolean expressions and c the set of program expressions (also called commands or statements). The syntaxes of e, b and c are given by the following extended BNF definition:

$$E ::= n \mid x \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1/E_2 \mid \ldots$$
$$B ::= \textbf{true} \mid \textbf{false} \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid B_1 = B_2 \mid B_1 < B_2 \mid \ldots$$
$$C ::= \textbf{skip} \mid \textbf{assert}(B) \mid x := E \mid a[E_1] := E_2 \mid C_1; C_2 \mid \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{while } B \textbf{ do } C \mid \ldots$$

The ellipses above mean that other forms of integer expressions could be added according need. It is also important to note that this language does not consider indentation, so we will use that advantage to make reading easier below. Finally, assume, for the rules in this subsubsection, that the two given programs C_1 and C_2 are separable.

To derive valid Hoare quadruples, there are axioms and inference rules that reason about the relation between two programs [17]. We can divide these rules in at least three

sets that increment the former set with some additional rules, meaning that 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 C_1 and C_2 must have the same shape and execute in lockstep. This creates a situation where we can not derive a simple Hoare quadruple like $\{\Phi\}\ C; \textbf{skip} \sim C\ \{\Psi\}$. To allow this and other important derivations, we present next other 4 rules that enable the separate reasoning of C_1 and C_2. Together with the rules of the simplest set, these rules constitute the base set:

The base set of rules of RHL is incomplete, since it does not provide a way to reason about programs whose loop structures differ or, even when they do not, the pre-condition is not strong enough to prove that the loops execute in lockstep. This problem can be overcome when we the extend the base set with one more rule: self-composition.

Note that the premise of this rule is not a Hoare quadruple but a Hoare triple since there is no $\sim$ operator, only a sequence of C_1 and C_2 within a single program. This triple can also be presented as the following quadruple:

$$\{\Phi \wedge \Phi'\}\ C_1; C_2\ \sim\ C_1'; C_2'\ \{\Psi \wedge \Psi'\}$$

If we keep the programs $C_1; C_2$ and $C_1'; C_2'$ *separable* by renaming their variables and if $\Phi = \Phi'$ and $\Psi = \Psi'$, we get to the self-composition rule presented before. Another way of looking at self-composition is that it reduces a Hoare quadruple into a Hoare triple by transforming the two related programs C_1 and C_2 into a new one that executes them

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

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

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

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

Figure 2.2: Base set of rules of RHL.

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

Figure 2.3: Extended set of rules of RHL.

in sequence ($C_1; C_2$). Finally, it is important to mention that relative completeness of the simplest or base sets of rules of RHL extended with self-composition follows immediately from relative completeness of self-composition [17].

2.2 OCaml

2.2.1 History & Characteristics

OCaml [9] 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 [18], released in 2014 and they have been expanding since. Kotlin, firstly released in 2016, was already designed with the functional style in mind [19].

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 [20]. But there is more.

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

1. Alt-Ergo [21], a popular SMT solver that has presence in Cameleer;
2. Coq [22], a formal proof management system;
3. and even the web version of Facebook Messenger [23]!

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)
```

OCaml
```
let gcd_iter (a0: int) (b0: int) : int =
    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"* [13]. 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.

When we want to verify the correctness of a given program, we need to ensure that the input code is written in WhyML [24], since that is the language that Why3 expects. WhyML can be considered a ML dialect and is simultaneously a specification and a programming language that serves as an intermediary language between C, Java or Ada programs and the verification conditions (VCs) generated by Why3.

The most used WhyML specification keywords are precisely the ones present in the example in the next subsection, *requires* and *ensures*. These represent the pre-conditions and post-conditions, respectively. Besides those, the *variant* and *invariant* keywords are also very common. Loops in this language are annotated with invariants, through the use of the *invariant* keyword. While loops specifically and recursive functions benefit greatly from the *variant* keyword, which represents a way of showing that the program does not run forever.

2.3.1 Example

As a way of demonstrating how Why3 works in practice, here is a preview of what is possible to do with product programs, an important concept which we will discuss in detail in the next chapter. If we want to prove that the original and the transformed programs are equivalent, we can write a third program that combines the first two, a *product program*.

The original program is based on only two assignment instructions: *y := x + 1;* and *z := y + 1;*. To make the programs separable, we renamed *x*, *y* and *z* in the original program

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 2.4: Program based on assignments.

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 2.1: Program based on assignments verification results.

2.4 GOSPEL

GOSPEL [11] (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:

OCaml + GOSPEL

```
(*@ 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 *)
```

OCaml + GOSPEL

```
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 time 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?

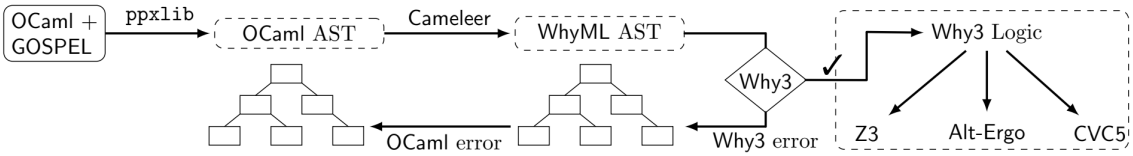

Figure 2.5: Architecture and verification pipeline, taken from [12].

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

2.5.1 Example

We will now demonstrate the execution of the verification of a GOSPEL annotated OCaml program, using the code presented in the previous section. To start the verification, assuming the code is saved in a file called **gcd.ml**, we use the following command: **cameleer gcd.ml**. This will feed our code to the Cameleer pipeline and, even if there are errors, a new window with the Why3 GUI will appear.

Now, we can use the *Split VC* strategy (Tools -> Strategies -> Split VC) to separate our bigger goal into several, smaller and simpler VCs, for each implementation. In this case, if we use the *Auto level 0* strategy (Tools -> Strategies -> Auto level 0) on the **gcd.ml** goal, our proof successfully terminates.

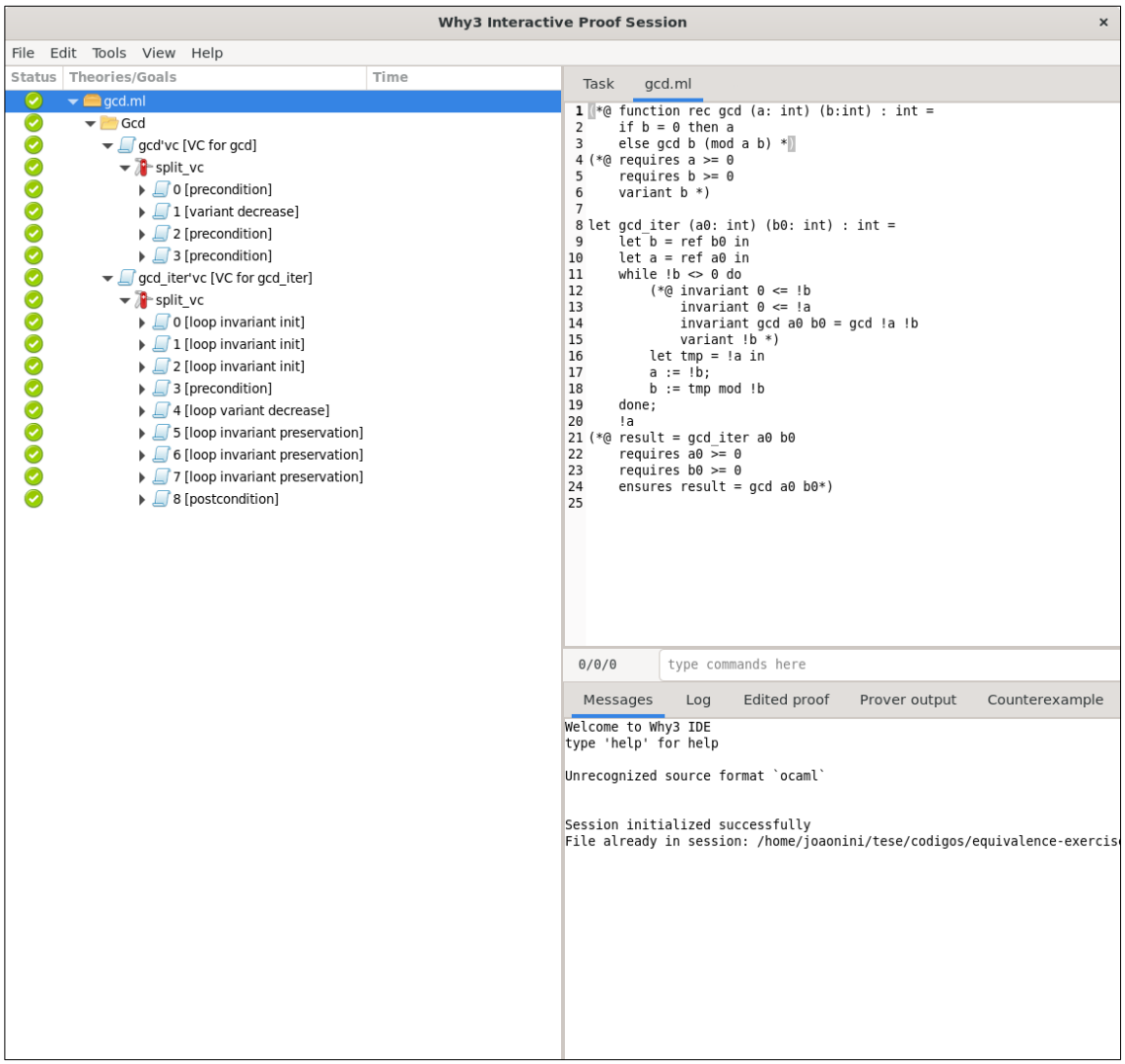

Figure 2.6: Verifying **gcd.ml** with the Why3 IDE.

You can now get automatically generated Latex code that shows how much time each VC took to be proven, using the command *why3 session latex -o proof_tex test*. Do not forget to create the proof_tex directory, otherwise you will get an error. The Latex code for this

example will look like the following. As you can see, CVC5 was the prover used and it took double the time to discharge the imperative implementation (0.46 seconds) than its functional counterpart (0.23 seconds).

Proof obligations		CVC5 1.0.6
lemma `VC for gcd`	lemma `precondition`	0.04
	lemma `variant decrease`	0.08
	lemma `precondition`	0.03
	lemma `precondition`	0.08
lemma `VC for gcd_iter`	lemma `loop invariant init`	0.03
	lemma `loop invariant init`	0.04
	lemma `loop invariant init`	0.04
	lemma `precondition`	0.04
	lemma `loop variant decrease`	0.08
	lemma `loop invariant preservation`	0.08
	lemma `loop invariant preservation`	0.05
	lemma `loop invariant preservation`	0.05
	lemma `postcondition`	0.05

3

State of the Art

Definition: Structural equivalence - we say that two programs are structurally equivalent if they present the same sequence of statements/commands. For example, a program P_1 that consists in two assignments and a while loop is not equivalent to another program program P_2 if the latter's structure is a single for loop. On the other hand, if P_1 and P_2 both contain, let us say, three assignments followed by a for loop (with a single assignment inside) that executes the same number of iterations, they are structurally equivalent, despite the values being assigned to the variables.

Note that this definition is useful when reading about self-composition and cross-products below in this capter. However, we aim to allow verification of programs that would not be considered by this definition of structural equivalence and therefore be more flexible, but we still need some degree of similarity. That is why we do not target programs that include exceptions.

3.1 Self-composition

3.1.1 Introduction

The proposal of self-composition [25] is 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 tried 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 specified before 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.

3.1.2 Exploring non-interference

In order to give an example of self-composition in the next subsection, we present now a simple program that respects the properties of termination-insensitivity and non-interference.

Consider a simple, deterministic and imperative language that allows sequential composition and the evaluation relation $\langle P, u\rangle \Downarrow v$, where P is a program and u, v are memories. Memory in this context means a map where the keys are the program variables of P and the values are the values of those program variables. Furthermore, consider that all variables of P need to be either public or private, let $\overrightarrow{x}$ be the set of all public variables in P and $\overrightarrow{y}$ the set of all of its private variables. For all memories u_1, u_2, v_1, v_2, the properties of termination-insensitivity and non-interference for P can be described as:

$$[\langle P, u_1\rangle \Downarrow v_1 \ \wedge \ \langle P, u_2\rangle \Downarrow v_2 \ \wedge \ u_1 =_L u_2] \Rightarrow v_1 =_L v_2$$

where $=_L$ is the point-wise extension of equality on values to public parts of memories.

Consider $[\overrightarrow{x'}, \overrightarrow{y'}/\overrightarrow{x}, \overrightarrow{y}]$ as being a renaming of the program variables $\overrightarrow{x}$ and $\overrightarrow{y}$ of P with fresh variables $\overrightarrow{x'}$, $\overrightarrow{y'}$ and let P' be as P but with its variables renamed: $P[\overrightarrow{x'}, \overrightarrow{y'}/\overrightarrow{x}, \overrightarrow{y}]$. Also, to refer to the the disjoint union of two memories we will use the $\uplus$ symbol. Therefore, we have $\langle P, u\rangle \Downarrow v \ \wedge \ \langle P', u'\rangle \Downarrow v'$ iff $\langle P; P, u \uplus u'\rangle \Downarrow v \uplus v'$. We can now rearrange non-interference, for memories u, u', v, v', as the following:

$$(\langle P; P, u \uplus u'\rangle \Downarrow v \uplus v' \ \wedge \ u =_{\overrightarrow{x}} u' \circ [\overrightarrow{x}/\overrightarrow{x'}]) \Rightarrow v =_{\overrightarrow{x}} v' \circ [\overrightarrow{x}/\overrightarrow{x'}]$$

where $\circ$ is the symbol for function composition and $=_{\overrightarrow{x}}$ is the point-wise extension of equality on values to the restriction of memories to $\overrightarrow{x}$. With this formulation, we are able to reduce the non-interference property of P to a property of all the executions of $P; P'$. Therefore, an alternative characterization of non-interference can be presented by using programming logics, since they are sound and relatively complete with relation to the operational semantic. We can describe non-interference using Hoare triples:

$$\{\overrightarrow{x} = \overrightarrow{x'}\}\ P; P'\ \{\overrightarrow{x} = \overrightarrow{x'}\}$$

3.1.3 Code Example

Consider the following program:

$$x := y;\ x := 0$$

and $x \mapsto x'$ and $y \mapsto y'$ as the renaming functions. The program is non-interferent iff

$$\{x = x'\}\ x := y; x := 0\ x' := y'; x' := 0\ \{x = x'\}$$

We are able to characterize information flow policies, which include some types of declassification, through the replacement of the =- relation by other (partial) equivalence

relation. More generally, this form of characterization allows us to use existing verification tools to prove (or disprove) information flow policies for a given program.

3.2 Cross-products

3.2.1 Introduction

Cross-products [26] aim to prove program equivalence, with a strong focus on verifying compiler optimizations. Instead of proving that the two input programs are equivalent, the analysis is done by combining them into one: the cross-product. With this, 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 of 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 [27] using CVC3 [28].

The results presented on the cited work reveal that CoVaC was able to verify a very considerable set of optimizations of LLVM [29], a complex modern compiler. Yet, the tool does not support verification of interprocedural optimizations or information flow policies, for example, but these are limitations that the authors showed interest in addressing.

3.2.2 A practical example to simplify the theory

Consider the following example, composed by a source program and its optimized version:

Source Program (S)	**Optimized Program (T)**
entry	*entry*
read x	*read x*
y := x + 0	*z := x*
*z := y * 1*	*write z*
write z	*exit*
exit	

This example is small but enough to demonstrate the idea of cross-products in practice. The transformation removes trivial arithmetic that does not change the final result and

represents, actually, two optimizations: constant folding ($y := x + 0; z := y * 1$ becomes $y := x; z := y$) followed by dead code elimination ($y := x; z := y$ becomes $z := x$).

Let us now informally model each program as a small transition graph with nodes (locations) and labeled edges. We mark $read$ and $write$ as observable transitions, while assigns are internal but still visible to the cross-product as assignment labels. Consider the source program's (S) nodes: S_0 = entry, S_1 = after read, S_2 = after $y := x + 0$, S_3 = after $z := y * 1$, S_4 = exit. Therefore, the edges of S can be described as:

$(S_0 \rightarrow S_1)$: *read(x)*
$(S_1 \rightarrow S_2)$: $y := x + 0$
$(S_2 \rightarrow S_3)$: $z := y * 1$
$(S_3 \rightarrow S_4)$: *write(z)*

Consider now the optimized program's (T) nodes: T_0 = entry, T_1 = after read, T_2 = after $z := x$, T_3 = before write, T_4 = exit. T's edges are the following:

$(T_0 \rightarrow T_1)$: *read(x)*
$(T_1 \rightarrow T_2)$: $z := x$
$(T_2 \rightarrow T_3)$: ϵ
$(T_3 \rightarrow T_4)$: *write(z)*

The ϵ steps are inserted as needed to align executions, as one can see in the transition between T_2 and T_3. Next, we construct the comparison graph $C = S \boxtimes T$. The nodes of C are pairs of the type $< S_i, T_j >$ that start at $< S_0, T_0 >$. There are 3 types of edge pairing: we match reads with reads, writes with writes (and then check the written values) and assignments with assignments. Regarding assignments, we only match them when both sides are at corresponding cutpoints (a given node in the control-flow graph), otherwise we allow an assignment to pair with an ϵ on the other side, making it wait. Therefore, the constructed comparison graph C contains paths that execute the two programs in lockstep, allowing ϵ-padding where needed. We will now showcase this using the example.

When we start, at $< S_0, T_0 >$, we see that both programs have the same instruction, $read(x)$, so we add an edge labeled $(read(x); read(x))$ to $< S_1, T_1 >$. At $< S_1, T_1 >$, the next instruction of the source program is $y := x + 0$, while in the optimized program the next command is $z := x$. Since they are both assignments, we simply compose them as in the previous edge, resulting in the addition of an edge with the label *(y := x+0 ; z := x)* to $< S_2, T_2 >$. Note that even though the expressions differ syntactically, the edge-matching logic in the compose algorithm allows pairing assignment edges. Later, the invariant generator / solver will establish if they compute the same value or not. From $< S_2, T_2 >$, we get another assignment from the source program $z := y * 1$ but an ϵ from the optimized program. According to rules, we can pair the source assignment with an ϵ on the target, making it wait, so we add an edge labeled *(z := y * 1; ϵ)* to $< S_3, T_3 >$. Finally, from $< S_3, T_3 >$ there is a *write(z)* in both programs, which means we add the last edge, with the label *(write(z); write(z))*, to the exit pair $< S_4, T_4 >$.

We will now discuss the invariant network. Consider φ_n an assertion associated with a node n, an assertion network $\phi_C = \{\varphi_n : n \in \text{locations of } C\}$ and d a data state. For a program C, we say that an assertion network is $invariant$ if, for every state $< n; \overrightarrow{d} >$ ocurring in a computation, $d \vDash \varphi_n$. This means that d satisfies the corresponding assertion φ_n on every visit of a computation of node n. The key cutpoints are after reads and before writes.

Consider x_s the notation for the variable x in the source program and x_t in the optimized version. We now describe the invariants for the example we have been using. After the $read(x)$, we reach $\phi < S_1, T_1 >$, where we say $x_s = x_t$. Next, after the first assignment on each side, $\phi < S_2, T_2 >$, we want to assert that the variable that will become z has the same value in both programs. After composing the actual assignments, we can infer the following. From S we get $y_s := x_s + 0$, which can be simplified to $y_s := x_s$. Next, also from S: $z_s := y_s * 1$, we get $z_s := y_s$, and since $y_s := x_s$, we get $z_s := x_s$. We also get $z_t := x_t$ after we go from T_1 to T_2, therefore, at $< S_3, T_3 >$, we can assert $z_s := z_t$ before the write. Inductively, we can express $\phi < S_3, T_3 >$: $z_s := z_t \wedge$ *(other observational equalities)*. This is exactly the kind of invariant network the CoVaC framework expects, and the authors also suggest a number of invariant generation techniques that can be used to compute these assertions automatically.

One of the most important phases of CoVaC is the checking of the witness verification condition. At every write node, the tool produces a simple verification condition: the invariant at that node must imply that the written values are equal. In our example, at the paired write node $< S_3, T_3 >$ the witness VC is $\varphi_{<S_3,T_3>} \rightarrow (z_s = z_t)$. If the invariant network proves $z_s = z_t$ at that node, the write outputs are equal; since reads/writes are the only observable I/O, this shows stuttering-equivalent observable behavior. According to the main theorem in the aforementioned work, the existence of such witness comparison graph and valid witness VCs implies that the target is a correct translation of the source.

Intuitively, this approach proves correctness for the three following reasons. Firstly, the comparison graph represents *all aligned executions* of source and target, with ϵ padding to allow one side to do assignments while the other waits. Furthermore, the invariants guarantee that at observable points (reads, writes, and procedure boundaries) the observable variables agree. Finally, the witness VC at writes enforces equality of output values. Together, these satisfy cross-product's definition of a witness, therefore the translation is correct.

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 [16, 30] is the constraint of *structural equivalence*. Inversely, traditional program verification has diverse and extensive tool support. Therefore, as a way of getting around that limitation, we can take advatange of the existing tool support if we are able to soundly reduce relational verification tasks into standard verification ones. This means we would translate Hoare quadruples ($\{\Phi\}$ $P_1 \sim P_2$ $\{\Psi\}$) into Hoare triples ($\{\phi\}$ P $\{\psi\}$), making sure that: if the original quadruple is valid, then the generated triple is also valid; if the if the original quadruple is not valid, the generated triple would also be invalid. Considering $\vDash$ the symbol for validity, the objective is finding ϕ, P, ψ that:

$$\vDash \{\Phi\}\, P_1 \sim P_2\, \{\Psi\} \quad \rightarrow \quad \vDash \{\phi\}\, \mathbf{P}\, \{\psi\}$$

Let us consider two imperative and *separable* programs P_1 and P_2. This enables the capacity of the assertions to be seen as first-order formulas about the variables of P_1 and P_2. Self-composition [25] represents a way of establishing the wanted equalities mentioned above: $P \equiv P_1;P_2$, $\phi \equiv \Phi$ and $\psi \equiv \Psi$. Despite being characterized by soundness and relative completeness [17], this construction is impractical for the authors of this [31] work. One of the two main reasons 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 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 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 [10] 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. Product programs are a way to reduce relational verification into standard verification through the construction of a product program P that simulates the execution statements of any two programs P_1 and P_2.

3.3.2 From relational verification to standard verification

3.3.2.1 A first intuition

As a way to provide a first intuition of construction of a product from structurally dissimilar programs before we dive into the definitions and rules later in this subsection, consider the simple example below (assume $0 \leq N$).

The first step to build the product program is to unroll the first loop iteration of the source code before synchronizing the loop statements. Rather than relying plainly on self-composition, this idea maximizes synchronization and differs from a functional interpretation of the product components. Furthermore, this approach also reduces

Source program	**Transformed program**	**Product program**

`i := 1;`
`while (i ≤ N) do`
`  x += i;`
`  i++`
	`j := 1;`
	`while (j ≤ N) do`
	`  y += j;`
	`  j++`
		`i := 1; x += i; i++; j := 1;`
		`while (i ≤ N) do`
		`  x += i; i++;`
		`  y += j; j++`

Figure 3.1: First intuition source, transformed and product programs.

considerably the complexity of the invariants required to prove the product program. In the case of a sequential composition of the source and transformed programs, we would need to provide invariants of the form $x = X + \frac{i(i-1)}{2}$ and $y = Y + \frac{j(j-1)}{2}$, respectively, assuming the pre-conditions $x = X$ and $y = Y$. On the other hand, if we construct the product program, the loop invariant $i = j \wedge x = y$ is enough to verify that the two first programs above satisfy the pre and post-relation $x = y$.

3.3.2.2 Establishing the ground rules

The commands in our programming model will stick to the grammar defined previously here, but we redefine some of its components in a more specific way.

x ranges over a set of integer variables V_i, a ranges over a set of array variables V_a, let V denote $\mathrm{V}_i \cup \mathrm{V}_a$, and assuming $\mathrm{V}_i \cap \mathrm{V}_a = \emptyset$. Let $e \in$ AExp and $b \in$ BExp range over integer and boolean expressions. Their semantics are given by $(\![e]\!])_{e \in AExp} : \mathrm{S} \to \mathbb{Z}$ and $(\![b]\!])_{b \in BExp} : \mathrm{S} \to \mathbb{B}$, respectively. Execution states' representation is $S = (V_i \rightharpoonup \mathbb{Z})\ X\ (V_a \rightharpoonup (\mathbb{Z} \rightharpoonup \mathbb{Z}))$. The semantics of the commands are standard, deterministic and are based on the relation $\langle \mathrm{c}, \sigma \rangle \rightsquigarrow \langle \mathrm{c}', \sigma' \rangle$.

Notice that $\langle \textbf{skip}, \sigma \rangle$ marks the end of the programs, **assert**(b) blocks the execution of the program if b is false, and we let $\langle \mathrm{c}, \sigma \rangle \Downarrow \sigma'$ mean $\langle \mathrm{c}, \sigma \rangle \rightsquigarrow^* \langle \textbf{skip}, \sigma' \rangle$. An assertion ϕ is a first-order formula with variables in V. We let $[\![\phi]\!]$ denote the set of states satisfying ϕ. Finally, we let $\mathrm{var}(c) \subseteq \mathrm{V}$ and $\mathrm{var}(\phi) \subseteq \mathrm{V}$ denote the set of (free) variables of a command c and assertion ϕ, respectively.

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:

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

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 ($\vDash \{\phi\}\ c\ \{\psi\}$) 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:

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

3.3.2.3 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 increment 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 3.2: 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**($b_1 \Leftrightarrow b_2$) 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 $\vDash \phi\ c\ \psi$ then $\vDash \phi\ c_1 \sim c_2\ \psi$.*

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 3.3: 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 $\vDash \phi\ c_1 \sim c_2\ \psi$.*

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;
b̄[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++;
  b̄[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 3.4: 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 3.5: Induction variable strength reduction.

3.4 WhyRel

3.4.1 Introduction

WhyRel [8] is a tool that aims to verify relational properties of pointer programs based on relational region logic, in an auto-active way. It also supports state-based encapsulation and dynamic framing. This tool appears in a context where two types of tools exist but the differences between them is quite relevant. One of the types of tools are the ones that do not support well the relational verification of programs that dynamically allocate mutable state; these take much weight off users' shoulders but are only useful to a restrict range of programs. The other type of tools are the auto-active ones; these enable the verification of a considerably larger set of programs but require more user input. WhyRel is implemented in OCaml and relies on a library provided by Why3 for constructing and pretty-printing WhyML parse trees. This tool was evaluated by its authors through several case studies that demonstrate the effectiveness of relational region logic for alignment, expressing heap relations and relational reasoning that exploits encapsulation.

WhyRel is the most important practical work for this thesis. Nevertheless, we will purposefully omit a considerable part of the details about WhyRel, since the verification of pointer programs is out of the scope of this thesis. Therefore, the focus will be directed towards functional programs.

3.4.2 Design details

Some of the most relevant relational properties are conditional program equivalence, non-interference and sensitivity [32]. WhyRel addresses tooling that enables modular verification of relational properties of heap-manipulating programs, including the ones that act on differing data representations and involve dynamically allocated pointer structures. It reaches modular reasoning about programs that contain pointers through

local reasoning, using frame conditions and procedural and data abstraction. Simpler relational invariants and specs can be achieved through the alignment of intermediate execution steps, a type of compositionality.

There are two approaches to relational properties verification supported by WhyRel. One of them corresponds to the reduction of the verification to the proof of functional properties of the two source programs, but this method suffers from two problems. It does not adapt well to complex programs and relational properties; it also prevents us to exploit similarities between the programs involved or reason through relational specs in a modular way. The second technique uses appropriate program alignments to prove the desired relational property. WhyRel uses biprograms to represent those alignments, that point out similarities between the two programs so that we can reason about their effects jointly. Relational properties of a given alignment entail the corresponding relation between the source programs, in case the chosen alignment is adequate, i.e., it captures all pairs of executions of the two underlying programs.

WhyRel invests a lot in encapsulation, as a way to hide internal representation details from clients and to verify the modular correctness of a client independently of its internal organization. Since this challenging on a technical level, WhyRel specifies encapsulation as a *dynamic boundary*, a kind of dynamic frame. A dynamic boundary captures the set of the module's internal locations. Therefore, enforcing encapsulation in practice requires making sure that clients do not modify any internal locations of a module. Relational logic has thorough soundness proofs [33] and WhyRel, according to its authors, is an accurate implementation.

This tool should be interpreted as a front-end to Why3, where users provide the code, specifications and annotations and, additionally for relational verification, they should also provide relational specifications and alignments using a syntax for product programs. After that, WhyRel translates these inputs into WhyML code, enconding them in a way that accurately captures the representation of the heap model and fine-grained framing formalized in relational region logic. Finally, some VCs are added by WhyRel as intermediate assertions and lemmas for the user to confirm, and the verification itself is achieved mainly through the use of the Why3 IDE.

3.4.3 Patterns of program alignment

Choosing alignments to decompose relational verification is an important task since it directly impacts how simple the relational assertions and loop invariants can be to prove the equivalence of the programs in question. In this subsection, we present two examples of biprograms that capture alignments that are not maximal. We will explain what that means below.

3.4.3.1 Differing control structures

```
meth mult1(n: int, m: int) : int =
  i := 0;

  while (i < n) do
    j := 0;

    while (j < m) do
      result := result + 1;
      j := j + 1;
    done;

    i := i + 1;
  done;
```

```
meth mult2(n: int, m: int) : int =
  i := 0;

  while (i < n) do
    result := result + m;
    i := i + 1;
  done;
```

Figure 3.6: Two different programs that multiply two integer numbers.

In this [34] work, the authors developed a way to establish program equivalence through state-dependent alignments of program traces. They identified a challenge when trying to confirm the equivalence of two programs that multiply two non-negative integer numbers using different control flow, as shown in the figure. The challenge arises in the context of automated approaches to relational verification specifically, because of the necessity of aligning an unbounded number m of loop iterations on the left (*mult1*) with a single iteration on the right (*mult2*).

```
meth mult_bip (n: int, m: int | n: int, m: int) : (int | int) =
  ⌊ i := 0 ⌋;

  while (i < n) | (i < n) do
    invariant { i ≐ i ∧ result ≐ result }

    (
      j := 0;
      while (j < m) do
        result := result + 1;
        j := j + 1;
      done
      |
      result := result + m;
    )

    assert { ◁ result = old(result) + m ◁ };
    ⌊ i := i + 1 ⌋;
  done;
```

Figure 3.7: An example of a biprogram for the example in this figure.

In order to prove equivalence, we verify the biprogram presented in this figure. To achieve that, the pre-relation (or pre-condition) should be $n \doteq n \wedge m \doteq m$ and the

post-relation (or post-condition) $result \doteq result$. The $\doteq$ symbol means that the left side of the expression (only reasoning about the left program) is equal to the right side of the expression (this side also only reasons about the right program). Therefore, the specification above means that if the input variables have the same values for both programs, then the output will also be the same.

The | symbol means that the instructions on its left refer to the left program and the instructions at its right refer to the right program. The $\lfloor$ and $\rfloor$ symbols mean that the instruction inside is executed by both sides. This means that if we have $C_1|C_2$ and C_1 is exactly the same instruction as C_2, we can simply write $\lfloor C_1 \rfloor$.

We call unary programs the source programs, in this case, *mult1* and *mult2* present in this figure. We say that an alignment is maximal if every statement/command is the same for the two unary programs. This example's alignment is therefore not maximal, since we have two while loops on the left side and only one on the right side. Nevertheless, we can still explore their similarities through the alignment of the outer loops in lockstep, and the left inner loop with the *result := result+m;* instruction on the right.

Observe that the notation $\triangleleft F \triangleleft$ (respectively $\triangleright F \triangleright$) means that the unary formula F is true for the state of the left (respectively right) program. To prove that *mult1* and *mult2* are equivalent, we only need the invariant that shows equivalence on i and *result* for every iteration of the outer loop. Additionally, to prove this invariant, we need to state that the inner loop of the left side has the same effect as the single assignement on the right: incrementing the *result* by m. In this case, that reasoning comes to life through the assertion after the left inner loop and before the assigment to i. Notice that *old(result)* inside the assertion represents the value that the $result$ variable had in the previous loop iteration.

3.4.3.2 Conditionally aligned loops

The example presented in the previous subsubsection takes advantage of the fact that the two outer loops are executed the same number of times, which can be inferred from their conditions being the same ($i < n$). It allows a lockstep alignment of the loops' iterations, leading to simple relational invariants. However, we can not exploit this lockstep reasoning on all cases, forcing us to use other patterns of loop alignment.

Considering the example in the figure, observe that *ex1* calculates the factorial of x and *ex2* computes the result of 2^x, both for $x \geq 4$. In this case, we do not want to prove the equivalence of *ex1* and *ex2*, but that the factorial majorizes the exponent for $x \geq 4$. Therefore, we provide the following relational specification to reach that goal. Keep in mind that the post-condition, indicated by the *ensures* keyword, should be interpreted as "the value of the variable *result* of *ex1* is strictly greater than its *ex2*'s counterpart, after the execution ends".

This relational spec means that, if at the start of the execution: both programs *x's* are equal; *ex2's x* is greater or equal to 4; and both *n's* are strictly greater than 0, then the

```
meth ex1 (x: int, n: int) : int =
  y := x;
  z := 24; // 4!
  w := 0;

  while (y > 4) do
    if (w mod n = 0) then
      z := z * y;
      y := y - 1;
    end;

    w := w + 1;
  done;

  result := z;
```

```
meth ex2 (x: int, n: int) : int =
  y := x;
  z := 16; // 2^4
  w := 0;

  while (y > 4) do
    if (w mod n = 0) then
      z := z * 2;
      y := y - 1;
    end;

    w := w + 1;
  done;

  result := z;
```

```
meth ex_bip (x: int, n: int | x: int, n: int) : (int | int) =
  ⌊ y := x ⌋;
  (z := 24 | z := 16);
  ⌊ w := 0 ⌋;

  while (y > 4) | (y > 4) . ◁ w mod n ≠ 0 ◁ | ▷ w mod n ≠ 0 ▷ do
    invariant { ◁ z ◁ > ▷ z ▷ ∧ y ≐ y ∧ ▷ y ≥ 4 ▷ }

    if (w mod n = 0) | (w mod n = 0) then
      (z := z * y | z := z * 2);
      ⌊ y := y - 1 ⌋;
    end;

    ⌊ w := w + 1 ⌋;
  done;

  ⌊ result := z ⌋;
```

Figure 3.8: Programs that compute the factorial, the exponent of x ≥ 4 and a biprogram.

```
meth ex_bip_spec (x: int, n: int | x: int, n: int) : (int | int) =
  requires { x ≐ x ∧ ▷ x ≥ 4 ▷ ∧ Both (n > 0) }
  ensures { ◁ result ◁ > ▷ result ▷ }
```

Figure 3.9: Relational spec for the example in this figure.

output of the left program will be strictly greater than the output of the right side.

ex1 and *ex2* structure's are very similar and the example is simplified by setting the z to 4! and 2^4, respectively. The w variables' goal is to introduce *stuttering* steps. In this case, it restricts the loops to only execute the assignments to z and y when w is divisible by n, something that is valid for both sides.

Since the only restriction on n is that they both sould be greater than 0, we can not

be sure if the number of iterations will be the same for both sides, which excludes the possibility of providing an alignment based on pure lockstep. Therefore, we should only align iterations in lockstep when $w \ mod \ n = 0$ on both sides; in other words, when the assignments to z and y will be performed. For the other situations, when a side will not update those variables in the current iteration, the alignment of the iteration on the other side should be done with executing nothing, represented by the *skip* command. With these two types of alignments, we are able to establish the invariant $\triangleleft$ z $\triangleleft$ > $\triangleright$ z $\triangleright$.

This alignment is represented in terms of syntax by the *ex_bip* biprogram in this figure. Conditional alignment is captured using additional annotations, the *alignment guards*. These annotations are general relation formulas that express conditions that cause the iterations to be either left-only, right-only or lockstep. In our example, the alignment guards are $\triangleleft \ w \ mod \ n \neq 0 \ \triangleleft \ | \ \triangleright \ w \ mod \ n \neq 0 \ \triangleright$. Looking at the left alignment guard, $\triangleleft \ w \ mod \ n \neq 0 \ \triangleleft$, we understand that *ex_bip* executes left-only iterations when, on the left side, w is **not** divisible by n. A left-only iteration means that only the left side statements inside the loop are executed: *if (w mod n = 0) then (z := z*y; y := y-1) end; w := w+1*. The only assignment in this case is the increment of the w variable, since the alignment guard is contraditory with the if's condition. This reasoning is similar to the right side, representing the right-only iterations. Finally, the lockstep iterations happen when both alignment guards are false; in this case, when w is a multiple of n on both sides. This alignment allows us to use the relational invariants in *ex_bip* to prove that the post-condition holds after the execution of the biprogram.

3.4.4 Translating biprograms into product programs

One of the main goals of WhyRel (also one of our main goals) is translating biprograms into product programs. This tool achieves that through WhyML functions that act on a pair of states; WhyML functions that encode biprograms also act on a refperm, renaming references allocated in the two states being related. However, before the translation phase, WhyRel checks how adequate the biprogram in question is. At an intuition level, we can think that if the two unary programs are related in accord with a relational spec, then the biprogram is adequate if it satisfies that spec, which suggests that relatedness strongly impacts the correctness of the biprogram.

In practice, WhyRel confirms the adequacy of the biprogram in two phases, by checking if it can cover all pairs of executions of its two source programs. The first phase is done by checking the syntax in order to ensure that the biprogram was really constructed from its underlying programs, through projection operations. Consider the following notation, where CC represents a biprogram, $\overleftarrow{CC}$ its left projection (the left unary program) and $\overrightarrow{CC}$ its right projection (the right unary program). As a simple example, the left projection of the following program is **x := 2; x := x + 1;** and the right projection is **x := x + 1;**.

```
( x := 2 | skip);
⌊ x := x + 1 ⌋;
```

Considering the unary original programs C and C', the corresponding biprogram CC and $\equiv$ the symbol for syntactic equality, WhyRel checks $\overleftarrow{CC} \equiv C$ and $\overrightarrow{CC} \equiv C'$. The second phase consists in the introduction of annotations in the biprogram, such as assertions or loop invariants, by WhyRel, with the objective of ensuring adequacy. In an hypothetical case where a biprogram aligns two loops in lockstep, this tool would add, as relational loop invariant, the equivalence of the two loop guards.

1. $\beta[\![C \mid C']\!](\tau_l, \tau_r) \triangleq \mu[\![C]\!](\tau_l);\ \mu[\![C']\!](\tau_r)$

2. $\beta[\![\lfloor m(x \mid y) \rfloor]\!](\tau_l, \tau_r) \triangleq \phi(m)\big(\tau_l.\mathrm{st}, \tau_r.\mathrm{st}, \epsilon[\![x]\!](\tau_l), \epsilon[\![y]\!](\tau_r)\big)$

3. $\beta[\![\lfloor C \rfloor]\!](\tau_l, \tau_r) \triangleq \beta[\![C \mid C]\!](\tau_l, \tau_r)$

4. $\beta[\![CC;\ DD]\!](\tau_l, \tau_r) \triangleq \beta[\![CC]\!](\tau_l, \tau_r);\ \beta[\![DD]\!](\tau_l, \tau_r)$

5. $\beta[\![\texttt{var}\ x : T \mid x : T'\ \texttt{in}\ CC]\!](\tau_l, \tau_r) \triangleq \texttt{let}\ x_l = \mathrm{def}(T)\ \texttt{in}\ \texttt{let}\ x_r = \mathrm{def}(T')\ \texttt{in}\ \beta[\![CC]\!]\big([\tau_l \mid x : x_l], [\tau_r \mid x : x_r]\big)$

6. $\beta[\![\texttt{if}\ E \mid E'\ \texttt{then}\ CC\ \texttt{else}\ DD]\!](\tau_l, \tau_r) \triangleq \texttt{assert}\{\ \epsilon[\![E]\!](\tau_l) = \epsilon[\![E']\!](\tau_r)\ \};\ \texttt{if}\ \epsilon[\![E]\!](\tau_l)\ \texttt{then}\ \beta[\![CC]\!](\tau_l, \tau_r)\ \texttt{else}\ \beta[\![DD]\!](\tau_l, \tau_r)$

7. $\beta[\![\texttt{while}\ E \mid E'\ \texttt{do}\ CC]\!](\tau_l, \tau_r) \triangleq \texttt{while}\ \epsilon[\![E]\!](\tau_l)\ \texttt{do}\ \texttt{invariant}\ \{\ \epsilon[\![E]\!](\tau_l) = \epsilon[\![E']\!](\tau_r)\ \}\ \beta[\![CC]\!](\tau_l, \tau_r)$

8. $\beta[\![\texttt{while}\ E \mid E'.\ P \mid P'\ \texttt{do}\ CC]\!](\tau_l, \tau_r) \triangleq$

$\texttt{while}\ (\epsilon[\![E]\!](\tau_l)\ \vee\ \epsilon[\![E']\!](\tau_r))\ \texttt{do}\ \texttt{invariant}\ \{A\}$
$\quad \texttt{if}\ (\epsilon[\![E]\!](\tau_l) \wedge \mathrm{F}[\![P]\!](\tau_l, \tau_r))\ \texttt{then}\ \mu[\![\overleftarrow{CC}]\!](\tau_l)$
$\quad \texttt{else if}\ (\epsilon[\![E']\!](\tau_r) \wedge \mathrm{F}[\![P']\!](\tau_l, \tau_r))\ \texttt{then}\ \mu[\![\overrightarrow{CC}]\!](\tau_r)$
$\quad \texttt{else}\ \beta[\![CC]\!](\tau_l, \tau_r)$
where $A \equiv (\epsilon[\![E]\!](\tau_l) \wedge \mathrm{F}[\![P]\!](\tau_l, \tau_r)) \vee (\epsilon[\![E']\!](\tau_r) \wedge \mathrm{F}[\![P']\!](\tau_l, \tau_r)) \vee (\neg\epsilon[\![E]\!](\tau_l)\ \wedge\ \neg\epsilon[\![E']\!](\tau_r)) \vee (\epsilon[\![E]\!](\tau_l)\ \wedge\ \epsilon[\![E']\!](\tau_r))$

Figure 3.10: Rules of translation of biprograms.

The rules of translation of biprograms into product programs are listed in this figure. β receives a biprogram and a pair of contexts (τ_l, τ_r) and transforms it into a WhyML program. Contexts map WhyRel identifiers to WhyML identifiers and store information about the state parameters that the generated WhyML will receive. In a similar way, μ translates unary programs into WhyML programs, ϵ map expressions into WhyML expressions and F transforms a restricted set of relation formulas also into WhyML expressions. Unary programs are not restricted to operate on a disjoint set of variables

but, if they do not, during translation, WhyRel naturally has to rename those variables' names. Rule 5 performs this renaming, where context τ_l (and respectively τ_r) is extended with the renaming of x with its copy x_l (and respectively τ_r is extended with the binding $x : x\_r$).

Rule 1 sequentially composes the unary translations of C and C'. $\lfloor C \rfloor$ is simply syntactic sugar for $(C|C)$ that are translated by rule 3, with the expection of method calls. Aligning method calls that establish that the relational spec associated with the method allow procedure-modular reasoning about relational properties. Using the ϕ symbol to represent a global method context, WhyRel translates these calls to calls to the correct WhyML product program (rule 2).

Rule 4 represents that a sequence of commands in the biprogram is translated to also a sequence of commands in the product program. Control flow statements require the generation of more proof obligations: in the case of the *if...then...else*, in rule 6, it appears as a runtime assertion; in a lockstep aligned loop (rule 7), it takes the form of a loop invariant stating that the guards are in agreement. Regarding conditionally aligned loops, described by rule 8, the pattern shown by the alignment guards $P|P'$ are captured by the generated loop body. This means that if the left (resp. right) guard evalutes to true and P (resp. P') holds, a left-only (resp. right-only) iteration is executed; if not, a lockstep iteration is performed.

The adequacy of the biprogram in this case is guaranteed by the requirement that A must be an invariant. A states that the loop can either perform a one-sided iteration or a lockstep until both sides finish. The alignment guards P and P' can be any relational formula, in the context of relational region logic. Nonetheless, the encoding of conditionally aligned loops regards a conditional that branches on these alignment guards. In Why3, P and P' have to be restricted for this to function correctly; they can not contain quantifiers, for example. WhyRel supports several types of elements to be in the alignment guards: the commonly used boolean connectives, one-sided boolean expressions, one-sided points-to-assertions and, as referred several times before, agreement formulas.

3.4.4.1 Translation example

Recall the multiplication unary programs and their biprogram. We now present its corresponding product program, after applying the translation rules described before. In order to demonstrate what would be the expected output of the tool when we feed it the biprogram, we developed a simplified and more readable version of this example.

Firstly, we had to add the pre-conditions *n_l >= 0* and *m_l >= 0*, the two variants and the invariants for the inner loop, since Why3 was not able to prove correctness without these extra pieces of specification. *n_l >= 0* is needed because if *n_l < 0*, the program would wrongly return *0, 0* every time. The second additional pre-condition is necessary to stop $result\_r$ from eventually reaching its correct value (assuming n_r is positive) while $result\_l$'s value stays at 0 during the whole execution of the program. The *variant { n_l -*

```
module MultPP
  use int.Int
  use ref.Ref

  let mult_whyml (n_l m_l n_r m_r : int) : (result_l: int,
    result_r: int)
    requires { n_l = n_r && n_l >= 0 && m_l = m_r && m_l >= 0 }
    ensures  { result_l = result_r }
  =
    let i_l = ref 0 in
    let i_r = ref 0 in
    let result_l = ref 0 in
    let result_r = ref 0 in

    while !i_l < n_l do
      variant { n_l - !i_l }
      invariant { !i_l = !i_r && !result_l = !result_r }

      label Old in
      let j = ref 0 in

      while !j < m_l do
        variant { m_l - !j }
        invariant { 0 <= !j <= m_l && !result_l = !result_r + !j }

        result_l := !result_l + 1;
        j := !j + 1
      done;

      result_r := !result_r + m_r;

      assert { !result_l = (!result_l at Old) + m_l };
      i_l := !i_l + 1;
      i_r := !i_r + 1;
    done;

    return !result_l, !result_r
end
```

Figure 3.11: *mult* product program (WhyML).

!i_l } and variant { m_l - !j } are needed to prove termination of the outer and inner loops, respectively. The *invariant { 0 <= !j <= m_l && ! result_l = ! result_r + !j }* is also not dispensable, since Why3 is not able to reason automatically about how *result_l* varies during the inner loop.

Additionally, because *old(result)* in WhyRel's biprogram language means the value that the *result* variable had in the previous loop iteration, but in WhyML it represents the value in the previous function call, we had to adapt this when translating. The best solution we found is the use of labels to capture the state of program before the execution of the inner while loop, therefore allowing us to access the value of *result* in the previous

iteration. Finally, with this specification, Z3 was able to discharge all VCs automatically and the proof took 0.04 seconds to complete.

Proof obligations		Z3 4.14.1
lemma `VC for mult_pp`	lemma `loop invariant init`	0.00
	lemma `loop invariant init`	0.01
	lemma `loop variant decrease`	0.00
	lemma `loop invariant preservation`	0.01
	lemma `assertion`	0.01
	lemma `loop variant decrease`	0.00
	lemma `loop invariant preservation`	0.01
	lemma `postcondition`	0.00

Table 3.1: *mult_pp* verification results.

4

From Biprograms to OCaml – Methodology and Tool

4.1 From WhyRel to bip2ml

WhyRel describes a pratical approach to the verification of program equivalence based on product programs. It also provides a set of rules that guides the translation, in the case of WhyRel, from the biprograms language to WhyML. Its authors deeply explore the problems that pointer based programs bring to the table, but this is not an objetive of this thesis. Instead, we defined and formalized a programming language called *BipLang*. BipLang's main purpose is writing biprograms and was based on the biprograms language used in WhyRel's work. BipLang was designed to accept Gospel annotations and to be as similar as possible to OCaml in terms of syntax and semantics. We did that in order to allow programmers that are used to that language to verify the equivalence of their programs without needing to learn a new language or completely rewrite their code or specification. Then, we transpile the biprograms using the translation rules defined by WhyRel's authors to produce Gospel-annotated OCaml code.

4.2 bip2ml

4.2.1 Overview and Architecture

bip2ml is a transpiler that takes Gospel-annotated BipLang code and outputs Gospel-annotated OCaml code. This tool brings the contributions of WhyRel into the world of OCaml, Gospel and Cameleer. All of the work done on this thesis is available at `https://github.com/JoaoNini75/MasterThesis`, which includes bip2ml's code, this thesis' LateX code and other related material.

We now describe bip2ml's pipeline and how it integrates with Cameleer. The users are expected to start with two programs and aim to formally verify that they are equivalent. There may be some exceptions to this that we will discuss later in the case studies chapter, but this is our starting point and main objective. The first step requires the user to provide

the alignment of the two unary programs, writing a BipLang program with the necessary Gospel annotations. After that, the user feeds bip2ml that .bip file and it starts its work to produce the OCaml + Gospel program. The transpiler's output file can then be fed to Cameleer, which in turn will take care of the rest of the verification process, exactly as if we gave it an OCaml program that we wrote manually.

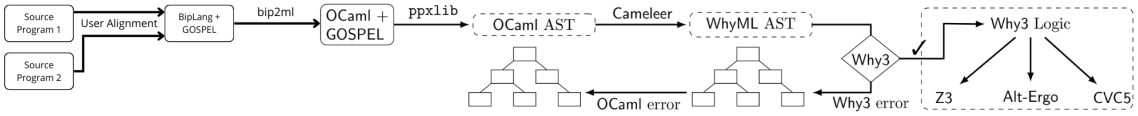

Figure 4.1: Architecture and verification pipeline of bip2ml and Cameleer.

The bip2ml transpiler is itself also a pipeline, that takes the input through four phases / components before terminating. The first component is the lexer, which defines what are the keywords and tokens accepted by BipLang; if any error occurs during this phase, bip2ml outputs *"lexical error"* to the terminal. We used ocamllex [35] to develop our lexer. Representing the second part of the transpiler, we have the parser, that takes the lexer's output and, by comparing it with the rules we define that are valid BipLang constructions, determines if the source code's structure is valid or not. If it is not, it outputs *"syntax error"* to the terminal. To program the parser, we utilized Menhir [36]. The output of the parser is an AST, the BipLang AST, which is the input of the next phase: the translator. This component is the one responsible for translating the BipLang AST into the OCaml AST, according to the 8 rules defined by WhyRel; this phase does not fail. Finally, we named the last part of our transpiler the printer, since its function is to receive the OCaml AST from the translator and outputting to the terminal or to a file the final OCaml code with all the Gospel specifications. Both the translator and the printer were implemented using OCaml directly.

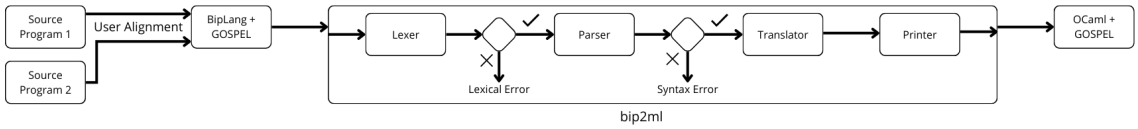

Figure 4.2: Architecture and pipeline of bip2ml.

bip2ml was tested with several examples that were separated into different files depending on the features they target. These tests were automated using Dune [37].

We also developed a VSCode extension that provides syntax highlighting for BipLang. Although it is dispensable, it certainly was extremely helpful during the whole development and testing of bip2ml, and it may also be a relevant aid to anyone that programs in BipLang. It was heavily based on OCaml Labs' VSCode OCaml Platform [38], since the syntaxes of OCaml and BipLang are nearly identical.

4.2.2 BipLang: language definition

BipLang's syntax, semantics and AST are very similar to OCaml's. Although BipLang implements the most common OCaml language constructions, there are definitely some missing, such as exceptions and object-oriented features. Taking that into consideration, it is also important to remember that, regarding the scope of this thesis, these constructions can not be seen as very relevant. On the other hand, our language adds 3 constructions that are not present in OCaml: the pipe, the floors and the conditionally aligned loops. Their semantics are the same as in WhyRel's work.

We now present the grammar of our language, BipLang. The syntaxes of the most relevant rules are given by the following extended BNF definition. Note that Biplang's keywords are written in orange.

```
file ::= decl*

decl ::=
          | def
          | spec
          | typedef

spec ::= { loc: location; text: string; }

typedef ::=
            | type id = payload (and typedef)?
            | type id = constructor (and typedef)?

payload ::=
            | bt
            | id

constructor ::= idc * payload?

def ::= let rec? id (parameter) : fun_ret? = expr spec

expr ::=
          | ()
          | (e)
          | c
          | id
          | -e
          | ¬ e
```

```
| ref e
| !e
| e + e
| e - e
| e * e
| e / e
| e mod e
| (e = e)
| e <> e
| e < e
| e ≤ e
| e > e
| e ≥ e
| e == e
| e != e
| e && e
| e || e
| e ^ e
| let id = e in e⃗
| if e then e else e
| for id = e to e do spec? e done
| while e do spec? e done
| id := e
| assert (e)
| match id with (case⃗)
| (e⃗)
| idc (e⃗)
| fun_name (e⃗)
| Module_name.fun_name (e⃗)
| def in e

arrays
| [| e⃗ |]
| id.(e)
| id.(e) <- e

lists
| [ e⃗ ]
| e :: [ e⃗ ]
| e @ [ e⃗ ]
```

```
        introduced by BipLang
        | e <|> e (pipe)
        | ⌊e⌋ (floors)
        | while e <|> e . e <|> e do spec? e done
        (conditionally aligned loops)

c ::=
        | None
        | int
        | bool
        | string
```

We simplified some things for the sake of readibility and there are some omissions in the grammar, which we will explain now. Firstly, we use c to simply represent a constant. Additionally, notice that we use id and idc. They both refer to identifiers but id refers to an all lowercase identifier, while idc stands for "capitalized id", so its first letter is capital. This is important to distinguish since module calls and type constructors, for example, use capitalized names, but lowercase identifiers are the standard for the names of variables and functions, among others. Next, we have to show the difference between bt (bip type) and at (any type). bt represents the basic types (bool, int, string, none), but at extends that definition to also allow id. This is useful, for example, when we have a parameter of a user-defined type:

```
type advanced_number =
        | Pos of int
        | Neg of int
        | Zero

let example (an : advanced_number) : advanced_number = begin
        (...)
end
```

Otherwise, we would be limited to parameters and function returns of the basic types (bt). $parameter$ is pretty straightforward: it represents a parameter, with all its possible variations, having pipe, floors or none, explicitly saying its type or not. fun_ret is optional and represents what the function returns, allowing explicit or implicit type definition and also pipe, floors or none of them. $constructor$ represents the definition of a constructor within a type defintion and $payload$ represents a type definition that has no constructors:

```
type constructors_example =
        | Pos of int
        | Neg of int
```

```
      | Zero

type payload_example = int * bool * string
```

case represents a case of the *match...with* construction when we do pattern matching. It is constituted by two parts, a pattern and an expression. The pattern represents the structure that each case will compare against the "parameter" of the *match...with* constructor, i.e, the left side of the arrow. The expression corresponds to what gets executed if a given case is the chosen one, i.e, the right side of the arrow:

```
let case_example (x : int) : int = begin
  match x with
  | 0 -> 10
  | 1 -> 11
  | _ -> -1
end
```

4.2.3 Translator and Printer

4.2.3.1 Design Choices and Implementation Details

Before we demonstrate how our translator and printer work by applying WhyRel's translation rules, there are some details about our printer that we want to make clear. Adding *_l* to the left side program and *_r* to the right side program was a printing choice that could be changed to anything the reader prefers when identifying to what program a given identifier belongs to. Additionally, it is relevant to mention that the output code you will read below was not formatted manually to be easier to read; it was only adjusted to fit the page. bip2ml effectively prints OCaml code respecting general good indentation practices. The choice of using two whitespaces for the indentation is also something that can easily change by simply setting a variable in our printer to a different value.

As a way to make the Gospel specifications coherent with the generated OCaml code and make sure which variable the user refers to, we require the user to write the specification using *_l* when reasoning about the left side and *_r* for the right side. Additionally, as a parser design choice, we also require that each function has a Gospel specification immediately after its declaration. If you do not want to write the specification in a given moment but want bip2ml to execute without a parsing error, you can simply do the following:

```
let your_function () = begin
  (your code...)
end
(*@*)
```

4.2.3.2 Demonstrating translation through examples

We now present 3 examples: the first applies rules 1, 2, 3 and 5, the second showcases rules 4, 6 and 7 and the third demonstrates rule 8. Note that we use, in our transpiler, the rules exactly as the authors of WhyRel defined them. We will omit the necessary Gospel specifications to prove these examples, since it is not our focus in this subsection. Notice that the names of the functions are different depending if they belong to the input or output; bip2ml does not apply this renaming, we did it to avoid confusion.

BipLang + GOSPEL

```
let invert_bip (x : int) = begin
  -x
end

let triple_bip (|_ s : string _|,
  n) = begin
  |_ s ^ s ^ s _|
end

let first_bip () = begin
  let number = 1 <|> 1 in
  let message = |_ "Hello" _| in
  let app_res = invert_bip (3) in
  let triple_res = |_ triple_bip
    (message, number) _| in
  message <|> message
end
```

OCaml + GOSPEL

```
let invert_ocaml (x : int) =
  -x

let triple_ocaml (s_l : string)
  (s_r : string) (n_l) (n_r) =
  (((s_l ^ s_l) ^ s_l),
   ((s_r ^ s_r) ^ s_r))

let first_ocaml () =
  let number_l = 1 in
  let number_r = 1 in
  let message_l = "Hello" in
  let message_r = "Hello" in
  let app_res = invert_bip (3) in
  let triple_res = triple_bip
    (message_l) (message_r)
    (number_l) (number_r) in
  (message_l, message_r)
```

Figure 4.3: Translation example for rules 1, 2, 3 and 5.

The first line of code of *first_bip*, *let number = 1 <|> 1 in*, uses rule 5 to rename the variables declared inside a *let...in*, with the same identifier and separated by a pipe. Therefore, in the OCaml version, we get the lines *let number_l = 1 in let number_r = 1 in*.

The next line, *let message = |_ "Hello" _| in*, makes use of rule number 3 to translate the expression inside the floors into a pipe with two equal expressions on each side. This shows that every floor is a pipe, but not every pipe can be a floor. This happens because if we have $C_1|C_2$ and $C_1 \neq C_2$, we cannot "compress" the pipe into a floored expression; otherwise, with $C_1 = C_2$, we would be able to establish that $C_1|C_2$ is the same as $\lfloor C_1 \rfloor$. So, bip2ml internally translates the original line to *let message = "Hello" <|> "Hello" in* and finally outputs *let message_l = "Hello" in let message_r = "Hello" in*, as we can see in the OCaml code.

The third line of *first_bip* is a binding to a function call: *let res = invert_bip (3) in*. In

this case, no rule is being used since the function call is not inside floors, therefore the input and output are the same. However, in the next line, *let triple_res = |_ triple_bip (message, number) _| in*, that is the case, so rule 2 is applied here. This rule interpolates the arguments of the function call so that we get the first argument for the left side, then that same argument for the right side, and all the other possible other ones, respecting the order defined in the biprogram and interpolating the left and right sides. So, we get the output *let triple_res = triple_bip (message_l) (message_r) (number_l) (number_r) in*.

Finally, in the last line of *first_bip*, *message <|> message*, we utilized rule 1, which translates two expressions separated by a pipe into two sequential expressions in general. In this particular case, however, since this is the value returned by the function, it returns them as a binary tuple, with the adequate identifier renaming: *(message_l, message_r)*.

BipLang + GOSPEL

```
let second_bip (|_c : bool _|,
  n : int) : |_int_| = begin

  if |_c_|
  then begin |_1_| end
  else begin
    let i = |_ ref 0 _| in
    let res = |_ ref 0 _| in

    while |_ !i < n _| do
      i := |_ !i + 1 _|;
      res := |_ !res * !i _|
    done;

    |_ !res _|
  end
end
```

OCaml + GOSPEL

```
let second_ocaml (c_l : bool)
  (c_r : bool) (n_l : int)
  (n_r : int) : int * int =

  assert ( (c_l) = (c_r) );
  if c_l
  then begin
    (1, 1)
  end else begin
    let i_l = ref (0) in
    let i_r = ref (0) in
    let res_l = ref (0) in
    let res_r = ref (0) in

    while (!i_l < n_l) do
      (*@ invariant ((!i_l < n_l))
                  ↔ ((!i_r < n_r))*)
      i_l := (!i_l + 1);
      i_r := (!i_r + 1);
      res_l := (!res_l * !i_l);
      res_r := (!res_r * !i_r)
    done;

    (!res_l, !res_r)
  end
```

Figure 4.4: Translation example for rules 4, 6 and 7.

Moving on to the second example, attent to the use of floors in the parameters and on the type returned by the *second_bip* function. The first statements indicate the use of an *if...then...else* block of code, which is translated by rule 6. Firstly, the rule introduces an

assertion that makes the program fail during its execution if the condition is not evaluated equally in both sides. After that, we use one of the conditions, in this case, WhyRel authors chose the left side one, but we could use the right side condition, since we just established that they are equal. The *then* and *else* branches will then get recursively translated, so we get this translated code: *assert ((c_l) = (c_r)); if c_l then begin (...) end else begin (...) end*. The *if* branch's translation is done by using rule 3 and then rule 1, returning that constant for both programs as a binary tuple: *(1, 1)*. Next, inside the *else* there is a while loop, which is translated following rule number 7. Similarly to the *if...then...else* block, we ignore the right side condition and use the left side one to limit the loop execution. However, contrary to the assertion used in rule 6, here a loop invariant is added so we guarantee that the condition on both sides is always evaluated to the same value during the whole execution of the *while*. The body of the cycle will then get recursively translated, which gets us the following translated code (and specification): *while (!i_l < n_l) do (*@ invariant ((!i_l < n_l)) <-> ((!i_r < n_r))*) (...) done*. Finally, we get to see rule 4 applied to the statements inside the while loop, translating sequential BipLang instructions to sequential OCaml instructions, applying the necessary transformations to those statements individually. The code lines we are mentioning are these: *i := |_ !i + 1 _| ; res := |_ !res * !i _|* . In practice, there is not a visual outcome of this rule, but, since we are here applying rule 3 too, the output looks like this: *i_l := (!i_l + 1); i_r := (!i_r + 1); res_l := (!res_l * !i_l); res_r := (!res_r * !i_r)*.

Let us look now at the third and last example. This one showcases the use of rule 8, that translates conditionally aligned loops. This rules takes as input a while loop with two alignment guards and outputs a while loop with a transformed body and a rather complex invariant, A.

A corresponds to the logical disjunction of 4 main formulas, that are all a conjunction themselves. The first represents the left loop condition *and* the left alignment guard. The second conjunction is the same but for the right side. The third corresponds to the conjunction of the negated loop conditions. The final conjunction is the same as the previous one but with the original loop conditions.

The generated invariant's structure is defined by the rule, varying only in the values that each expression is evaluated to. Nevertheless, our transpiler allows the user to add Gospel specifications to the original while loop and those reasonings will be copied and put together with the automatically generated A invariant, appearing in the output.

The body becomes a *if...then...else if...then...else* block, so there are three possible outcomes for each iteration of the cycle. The first occurs when the loop condition for the left program and the left alignment guard are both evaluated to true, therefore executing only the instructions that refer to the left side. The second branch works in a very similar way to the previous, but applied to the right side program. The third case happens when the loop condition (on the left side) is respected but none of the alignment guards are, resulting in the execution of all the instructions inside the original biprogram's cycle.

BipLang + GOSPEL

```
let counting_bip (|_ target : int _|, |_ mult_of _|) : |_ int _| = begin
  let counter = |_ ref 0 _| in

  while (!counter < target) <|> (!counter < target) .
        (!counter mod mult_of = 0) <|> (!counter mod mult_of = 0) do
    counter := |_ !counter + 1 _|
  done;

  |_ !counter _|
end
```

OCaml + GOSPEL

```
let counting_ocaml (target_l : int) (target_r : int) (mult_of_l)
  (mult_of_r) : int * int =

  let counter_l = ref (0) in
  let counter_r = ref (0) in

  while ((!counter_l < target_l) || (!counter_r < target_r)) do
    (*@ invariant
      (!counter_l < target_l && mod !counter_l mult_of_l = 0) ||
      (!counter_r < target_r && mod !counter_r mult_of_r = 0) ||
      (¬ (!counter_l < target_l) && ¬ (!counter_r < target_r)) ||
      (!counter_l < target_l && !counter_r < target_r) *)

    if ((!counter_l < target_l) && ((!counter_l mod mult_of_l) = 0))
    then begin
      counter_l := (!counter_l + 1)
    end else begin
      if ((!counter_r < target_r) && ((!counter_r mod mult_of_r) = 0))
      then begin
        counter_r := (!counter_r + 1)
      end else begin
        counter_l := (!counter_l + 1);
        counter_r := (!counter_r + 1)
      end
    end
  done;

  (!counter_l, !counter_r)
```

Figure 4.5: Translation example for rule 8.

5

Case Studies

In this chapter, we showcase several case studies that demonstrate the capabilities of the tool developed in this thesis. We start by presenting a simple example of what BipLang can do that OCaml cannot. Then, we focus on three case studies that are more complex and, therefore, demonstrate the real potential of bip2ml.

5.1 Incremeting OCaml

In this section, we give an example that showcases some of the functionalities and constructions that BipLang adds on top of OCaml. That is accomplished by the introduction of the ⟨|⟩ (pipe), ⌊ (left floor) and ⌋ (right floor) symbols. We also present our tool's outputs so we can compare the differences between writing each of these programs using BipLang versus direct OCaml.

Recall the *mult* unary programs and their biprogram. We also presented before its corresponding WhyML product program, the one that would be generated by WhyRel. Now, we rewrite the biprogram in BipLang and use our transpiler to get the generated OCaml code. Remember, the rules applied by bip2ml are the same as the ones developed by WhyRel's authors.

As described before, the *mult* biprogram combines two unary programs that multiply two non-negative integer numbers. On the left side, that is done through two nested while loops, incrementing the result by 1 in each iteration. On the right side, there is a single while loop with an assignment inside, making the result increment by m in each cycle iteration.

The specification of $mult\_biplang$ and $mult\_ocaml$ is the same as $mult\_whyml$, but now written in GOSPEL. Besides that, since OCaml does not support labels, we had to substitute it with a simple variable that is used later to get the value of the previous iteration. In practice, it also works, but WhyML's labels may help the readibility in terms of separating code and specification.

The reader may have also noticed that, in $mult\_ocaml$, bip2ml declares and initializes the j_r variable but it is never used. This can be seen as a limitation of our tool: we cannot

BipLang + GOSPEL

```
let mult_biplang (|_ n: int _|, |_ m: int _|) : |_ int _| = begin
  let i = |_ ref 0 _| in
  let res = |_ ref 0 _| in

  while |_ !i < n _| do
    (*@ variant   n_l - !i_l
        invariant !i_l = !i_r && !res_l = ! res_r *)

    let previous_res_l = !res_l in
    let j = |_ ref 0 _| in

    ((
      while (!j < m) do
        (*@ variant   m_l - !j_l
            invariant 0 ≤ !j_l ≤ m_l && !res_l = !res_r + !j_l *)
        res := !res + 1;
        j := !j + 1
      done
    )
    <|>
    (res := !res + m));

    assert ((!res_l = previous_res_l + m_l));
    i := |_ !i + 1 _|
  done;

  |_ !res _|
end
(*@ requires n_l = n_r && m_l = m_r && m_l ≥ 0
    ensures  match result with (l_res, r_res) → l_res = r_res *)
```

Figure 5.1: *mult* biprogram (BipLang).

mix "neutral" variables with "sided" ones. We also cannot mix left side and right side variables, but that would hardly make sense in the code, although it clearly makes sense in the specification. In this case, since the condition of the *mult_biplang*'s inner while loop is at the left of a *pipe*, those identifiers will be transformed into left side identifiers. So, even if we declared *let j = ref 0 in*, we would still get *while (!j_l < m_l) do*, and that would result in variables' names mismatching in the generated OCaml code.

Regarding the verification of *mult_ocaml*, CVC5 was able to discharge all VCs automatically in a total of 410 milliseconds.

OCaml + GOSPEL

```
let mult_ocaml (n_l : int) (n_r : int) (m_l : int) (m_r : int) :
  int * int =

  let i_l = ref (0) in
  let i_r = ref (0) in
  let res_l = ref (0) in
  let res_r = ref (0) in

  while (!i_l < n_l) do
    (*@ invariant ((!i_l < n_l)) ↔ ((!i_r < n_r))
        variant   n_l - !i_l
        invariant !i_l = !i_r && !res_l = ! res_r *)
    let previous_res_l = !res_l in
    let j_l = ref (0) in
    let j_r = ref (0) in

    while (!j_l < m_l) do
      (*@ variant   m_l - !j_l
          invariant 0 ≤ !j_l && !j_l ≤ m_l && !res_l = !res_r + !j_l *)
      res_l := (!res_l + 1);
      j_l := (!j_l + 1)
    done;

    res_r := (!res_r + m_r);
    assert ((!res_l = (previous_res_l + m_l)));
    i_l := (!i_l + 1);
    i_r := (!i_r + 1)
  done;

  (!res_l, !res_r)
(*@ requires n_l = n_r && m_l = m_r && m_l ≥ 0
    ensures  match result with (l_res, r_res) → l_res = r_res *)
```

Figure 5.2: *mult* product program (OCaml).

Proof obligations		CVC5 1.0.6
lemma VC for mult_ocaml	lemma loop invariant init	0.05
	lemma loop invariant init	0.04
	lemma loop invariant init	0.02
	lemma loop variant decrease	0.03
	lemma loop invariant preservation	0.04
	lemma assertion	0.06
	lemma loop variant decrease	0.04
	lemma loop invariant preservation	0.04
	lemma loop invariant preservation	0.05
	lemma postcondition	0.04

Table 5.1: *mult* product program (OCaml) verification results.

5.2 Real World Cases

This section illustrates the applicability of our tool through some more complex examples. As we did in the previous section, we present these programs written in BipLang and their corresponding translations to OCaml. The first and second examples correspond to the proof of two common compiler optimizations; while the first establishes the equivalence of the outputs of the source and optimized versions, the second case proves that a given part of the memory of the programs are in the same state when the execution finishes. The third example is not about program equivalence, but a similar relational property: majorization. These case studies show that bip2ml is able to reason about not only program equivalence, but other interesting properties too.

5.2.1 Induction variable strength reduction

The first example is a common compiler optimization, the induction variable strength reduction. Recall the source, optimized and product programs, presented earlier. Consider now the BipLang and translated OCaml programs.

We start by initializing the i, j and x variables. We use the floors for i and x, since both sides of the program bind to them the same value (*ref 0*, for both variables). To initialize j, on the other hand, we use the pipe, since one side binds to it *ref 0* and the other *ref c*.

We then have the main part of the program: the floored while loop. The condition of the cycle indicates that as long as $!i < n$ is true for both left and right, it keeps executing. This gets translated to only the left side condition and the first invariant in the loop of the generated program, automatically introduced by our tool. The other invariants and the variant had to be added by us. Then, inside the loop, we have two assignments to different identifiers, which explains the slightly different syntax of the form *id1 := value1 < | > id2 :=*

BipLang + GOSPEL

```
let induc_var_strength_red_biplang (|_b : int_|,
  |_c : int_|, |_n : int_|) : |_int_| = begin

  let i = |_ ref 0 _| in
  let j = ref 0 <|> ref c in
  let x = |_ ref 0 _| in

  while |_ !i < n _| do
    (*@ variant   n_l - !i_l
        invariant !i_l = !i_r
        invariant !x_l = !x_r
        invariant !j_r = !i_r * b_r + c_r *)
    j := !i * b + c <|> x := !x + !j;
    x := !x + !j    <|> j := !j + b;
    i := |_ !i + 1 _|
  done;

  |_ !x _|
end
(*@ requires b_l = b_r && c_l = c_r && n_l = n_r
    ensures  match result with (l_res, r_res) → l_res = r_res *)
```

Figure 5.3: Induction variable strength reduction (BipLang).

value2. The last instruction of the while is an equal assign to both sides of the program.

Finally, we return the value inside the x reference, for the left and right sides equally, which gets translated to the return of a binary tuple.

The pre-conditions and post-conditions are also provided by us. In this case, we are simply saying that agreement on the input results in agreement of the output. Let us look again at the loop's manually added invariants and variant. Since the assignments to x and i are the same for both sides, the invariants *!i_l = !i_r* and *!x_l = !x_r* are trivial, and so is the variant. The last invariant may be the most interesting one, but it is also not complex. Looking at the right side, each iteration (which represents an increment of *i_r* by 1), we add *b_r* to the previous *j_r*, and that corresponds to the multiplication of *i_r* by *b_r*. The + *c_r* comes from the fact that *j_r* starts at *c_r*, contrarily to the left side.

Regarding the proof duration, CVC5 was able to discharge all VCs automatically in 0.3 seconds.

OCaml + GOSPEL

```
let induc_var_strength_red_ocaml
  (b_l : int) (b_r : int) (c_l : int) (c_r : int)
  (n_l : int) (n_r : int) : int * int =

  let i_l = ref (0) in
  let i_r = ref (0) in
  let j_l = ref (0) in
  let j_r = ref (c_r) in
  let x_l = ref (0) in
  let x_r = ref (0) in

  while (!i_l < n_l) do
    (*@ invariant ((!i_l < n_l)) ↔ ((!i_r < n_r))
        variant   n_l - !i_l
        invariant !i_l = !i_r
        invariant !x_l = !x_r
        invariant !j_r = !i_r * b_r + c_r *)
    j_l := ((!i_l * b_l) + c_l);
    x_r := (!x_r + !j_r);
    x_l := (!x_l + !j_l);
    j_r := (!j_r + b_r);
    i_l := (!i_l + 1);
    i_r := (!i_r + 1)
  done;

  (!x_l, !x_r)
(*@ requires b_l = b_r && c_l = c_r && n_l = n_r
    ensures  match result with (l_res, r_res) → l_res = r_res *)
```

Figure 5.4: Induction variable strength reduction (OCaml).

Proof obligations		CVC5 1.0.6
lemma VC for induc_var_strength_red	lemma loop invariant init	0.03
	lemma loop invariant init	0.03
	lemma loop invariant init	0.03
	lemma loop invariant init	0.01
	lemma loop variant decrease	0.05
	lemma loop invariant preservation	0.03
	lemma loop invariant preservation	0.03
	lemma loop invariant preservation	0.03
	lemma loop invariant preservation	0.02
	lemma postcondition	0.04

Table 5.2: Induction variable strength reduction (OCaml) verification results.

5.2.2 Loop alignment

The second case study is about another compiler optimization known as the loop alignment. Contrarily to the previous example, this one is not exactly a proof that the programs' output are the same for the same input. In this case, we prove that, after the execution, a part of the memory of both programs is equal. Recall the source, optimized and product programs, presented earlier.

This biprogram and product program are slightly different from the product program presented before, but the proof was inspired by the one its authors developed; that one is available here. We renamed the j and renamed it to i_r, since it represents the same as i_l.

Regarding the pre-conditions, we require n_l to be equal or greater than 1 and both sides' n value should match too. There are restrictions on the the arrays' lengths, which should all be equal, between left and right side and also equal to $n\_l + 1$. Furthermore, the elements of both sides' a array should be in agreement, one by one; however, this requirement does not apply to the last element. The last pre-condition simply requires that the value of the first element of both b's to be the same. Finally, the post-condition establishes that both programs' memory finish in a state where the values of the elements of the d array match, excluding the first and last elements from this guarantee.

The program starts with an initial assignment and the unroll of the first loop iteration so we can reason about both programs in tandem during the cycle. Since the conditions of the loop are not exactly the same for both sides, we take advantage of rule 7 through a pipe, which generates the first invariant automatically: *invariant (!i_l < n_l) ↔ (!i_r < n_r - 1)*. Inside the while, there are three assignments, two that are exactly the same for both sides and one that differs and represents precisely the optimization. Finally, since the right side program is one iteration behind its left counterpart, we make one last assignment after the loop: *b_r.(n_r) ← a_r.(n_r)*.

Regarding the specification of the loop, we have a trivial variant that proves termination and 4 invariants. The first one, *invariant !i_r ≥ 0 && !i_l = !i_r + 1*, states that both i's are never negative and that the right side is always one iteration behind the left side, as we mentioned before. The second invariant, *b_l.(!i_r) = a_l.(!i_r)*, could be rewritten as *b_l.(!i_l + 1) = a_l.(!i_l + 1)* and simply states what happens in the first assignment of the cycle. Next, *invariant b_l.(!i_r - 1) = b_r.(!i_r - 1)* establishes that the last iteration assigned to both sides of the b array the same value, as we can see from the two first loop assignments and the pre-condition regarding the values of the elements of a. Finally, the invariant $\forall\, k.\ 1 \leq k < !i\_l \rightarrow d\_l.(k) = d\_r.(k)$ means that every position of the d array, on both sides, has the same value, starting at index 1 and going to the previous' iteration i.

The proof did not require any human interaction since CVC5 discharged all VCs in approximately 1.82 seconds.

BipLang + GOSPEL

```
let loop_alignment_biplang (|_ n : int _|, |_ a : int array _|,
  |_ b : int array _|, |_ d : int array _|) = begin

  let i = |_ ref 1 _| in
  assert (!i_l ≤ n_l);
  b_l.(!i_l) ← a_l.(!i_l);
  d_l.(!i_l) ← b_l.(!i_l - 1);
  i_l := !i_l + 1;
  d_r.(1) ← b_r.(0);

  while !i < n <|> !i < n - 1 do
    (*@ variant   n_l - !i_l
        invariant !i_r ≥ 0 && !i_l = !i_r + 1
        invariant b_l.(!i_r) = a_l.(!i_r)
        invariant b_l.(!i_r - 1) = b_r.(!i_r - 1)
        invariant ∀ k. 1 ≤ k < !i_l → d_l.(k) = d_r.(k) *)

    |_ b.(!i) ← a.(!i) _|;
    d.(!i) ← b.(!i - 1) <|> d.(!i + 1) ← b.(!i);
    i := |_ !i + 1 _|
  done;

  b_r.(n_r) ← a_r.(n_r)
end
(*@ requires n_l ≥ 1 && n_l = n_r
    requires Array.length a_l = n_l + 1
    requires Array.length b_l = n_l + 1
    requires Array.length d_l = n_l + 1

    requires Array.length a_l = Array.length a_r
    requires Array.length b_l = Array.length b_r
    requires Array.length d_l = Array.length d_r

    requires ∀ k. 0 ≤ k < n_l → a_l.(k) = a_r.(k)
    requires b_l.(0) = b_r.(0)

    ensures  ∀ k. 1 ≤ k < n_l → d_l.(k) = d_r.(k) *)
```

Figure 5.5: Loop alignment (BipLang).

OCaml + GOSPEL

```
let loop_alignment_ocaml (n_l : int) (n_r : int)
  (a_l : int array) (a_r : int array) (b_l : int array)
  (b_r : int array) (d_l : int array) (d_r : int array) =

  let i_l = ref (1) in
  let i_r = ref (1) in
  assert ((!i_l ≤ n_l));
  b_l.(!i_l) ← a_l.(!i_l);
  d_l.(!i_l) ← b_l.((!i_l - 1));
  i_l := (!i_l + 1);
  d_r.(1) ← b_r.(0);

  while (!i_l < n_l) do
    (*@ invariant (!i_l < n_l) ↔ (!i_r < n_r - 1)
        variant   n_l - !i_l
        invariant !i_r ≥ 0 && !i_l = !i_r + 1
        invariant b_l.(!i_r) = a_l.(!i_r)
        invariant b_l.(!i_r - 1) = b_r.(!i_r - 1)
        invariant ∀ k. 1 ≤ k < !i_l → d_l.(k) = d_r.(k) *)
    b_l.(!i_l) ← a_l.(!i_l);
    b_r.(!i_r) ← a_r.(!i_r);
    d_l.(!i_l) ← b_l.((!i_l - 1));
    d_r.((!i_r + 1)) ← b_r.(!i_r);
    i_l := (!i_l + 1);
    i_r := (!i_r + 1)
  done;

  b_r.(n_r) ← a_r.(n_r)
(*@ requires n_l ≥ 1 && n_l = n_r
    requires Array.length a_l = n_l + 1
    requires Array.length b_l = n_l + 1
    requires Array.length d_l = n_l + 1

    requires Array.length a_l = Array.length a_r
    requires Array.length b_l = Array.length b_r
    requires Array.length d_l = Array.length d_r

    requires ∀ k. 0 ≤ k < n_l → a_l.(k) = a_r.(k)
    requires b_l.(0) = b_r.(0)

    ensures  ∀ k. 1 ≤ k < n_l → d_l.(k) = d_r.(k) *)
```

Figure 5.6: Loop alignment (OCaml).

Proof obligations		CVC5 1.0.6
lemma VC for loop_alignment_biplang	lemma assertion	0.06
	lemma index in array bounds	0.04
	lemma precondition	0.05
	lemma index in array bounds	0.05
	lemma precondition	0.04
	lemma index in array bounds	0.05
	lemma precondition	0.05
	lemma loop invariant init	0.05
	lemma loop invariant init	0.02
	lemma loop invariant init	0.09
	lemma loop invariant init	0.10
	lemma loop invariant init	0.04
	lemma index in array bounds	0.03
	lemma precondition	0.05
	lemma index in array bounds	0.06
	lemma precondition	0.05
	lemma index in array bounds	0.06
	lemma precondition	0.06
	lemma index in array bounds	0.06
	lemma precondition	0.05
	lemma loop variant decrease	0.04
	lemma loop invariant preservation	0.04
	lemma loop invariant preservation	0.05
	lemma loop invariant preservation	0.03
	lemma loop invariant preservation	0.13
	lemma loop invariant preservation	0.33
	lemma index in array bounds	0.05
	lemma precondition	0.05
	lemma postcondition	0.04

Table 5.3: Loop alignment (OCaml) verification results.

5.2.3 Conditionally aligned loops

The third and last real world example is also not a proof of equivalence, but for a different reason. Instead, it is a proof that the left program's output is always larger than the right side program's output, assuming agreement of inputs and some other preconditions. Recall the unary programs, that compute the factorial and the exponent of $x >= 4$ and their biprogram. The *cond_align_loops_biplang* program is the same as WhyRel's biprogram, but rewritten in BipLang.

Our proof was inspired by the one developed by WhyRel's authors. Ours excludes the specification related to pointer-based programs, since that is not our target. However,

we include the *mult* axiom, which we found indispensable for the proof to finish without manually invoking tactics. Both our and WhyRel's proofs are partial, since we do not prove termination (as indicated by the *diverges* keyword). So, with no required human intervention, the proof was done by Z3 in approximately 0.32 seconds.

Now, if you look carefully at the translated program, *cond_align_loops_ocaml*, you may notice that there is dead code. Concretely, these *z_l := (!z_l * !y_l); y_l := (!y_l - 1)* and these *z_r := (!z_r * 2); y_r := (!y_r - 1)* branches. That code is unreachable since the loop condition and alignment guard, on the left side, get translated to *(!y_l > 4) && ((!w_l mod n_l) <> 0)*, and immediately after that we have an *if...then...else* block, in which the condition is contraditory *(!w_l mod n_l) = 0*. The same logic happens for the right side program. That is also the reason why the *invariant !y_l = !y_r* can be proven, since the only branch that updates these variables is the one that updates them at the same time (and with the same value). Nevertheless, this is simply a quirk of this specific example, that happens because the condition of the *if...then...else* block inside the while loop was defined as contraditory to the alignment guard by the authors of the program. Therefore, looking again at this other example of a conditionally aligned loop, you can see that there are no contraditory conditions that create cases of unreachable code.

Proof obligations		Z3 4.14.1
lemma VC for cal_ml	lemma loop invariant init	0.01
	lemma loop invariant init	0.01
	lemma loop invariant init	0.00
	lemma loop invariant init	0.01
	lemma precondition	0.00
	lemma precondition	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.00
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.00
	lemma loop invariant preservation	0.01
	lemma precondition	0.01
	lemma precondition	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.00
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma assertion	0.03
	lemma precondition	0.01
	lemma loop invariant preservation	0.02
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.02
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.00
	lemma loop invariant preservation	0.01
	lemma loop invariant preservation	0.01
	lemma postcondition	0.00

Table 5.4: Conditionally aligned loops (OCaml) verification results.

6

Conclusions

6.1 Contributions and Limitations

This thesis took the deep research done in the area of program equivalence, relational verification and product programs in order to produce a tool that helps programmers proof that two different programs are equivalent, among other relational properties. We created a language that can simultaneously be seen as a subset and a superset of OCaml, since it adds some symbols and constructions but does not have all of the features of OCaml. This happened because the focus of this work does not include pointer based programs, but all of the components of bip2ml were constructed with ease of extensability in mind. Therefore, if we come to realize that it makes sense, the necessary features could be added with relative ease. The similarities to OCaml are important since it means programmers that are comfortable with that language will have an easier adaptation period when writing code in BipLang. However, there are some restrictions imposed by the parser that make BipLang not as close to OCaml as it could be. These restrictions stem from technical difficulties that could not be tackled due to temporal limitations.

One of the main goals of our work was to bring the translation rules of WhyRel into the world of OCaml. This was possible due to the tight integration with GOSPEL and Cameleer. That enabled the reasoning about relational properties, through all of the bip2ml and Cameleer pipeline, to reach the powerful Why3 and all of its supported solvers and provers. Therefore, we were able to reduce relational verification into standard verification, which is currently a much easier path to prove program equivalence.

The case studies that we presented reveal that we were able to apply those rules correctly and use our tool to verify real world examples of program equivalence. Not only program equivalence, but also other relational properties such as majorization and therefore minorization as well. We were also able to establish the equivalence of two programs' memories after the execution of their product program finishes. Although we only showcased this for a part of the memories, we do not foresee obvious limitations to proving the equivalence of all the components of the memories. It would be only a matter of providing strong enough specification.

6.2 Future Work

In the future, this tool could very likely be used to reason about even more relational properties. It would be a matter of putting bip2ml to the test with more interesting examples. Those case studies could be complex compiler optimizations, **higher-order** constructs, **programs with heap memory**, **dynamic programming** and eventually **program evolution**. The **higher-order** constructs, like *iter* or *fold*, could be reasoned in tandem with other higher-order counterparts or even proving that a program with one of those constructs is equivalent to another that performs the same instructions with a *while* loop, for example. Regarding the **programs with heap memory**, we mean pointer based programs, which are already extensively discussed in the WhyRel work and that would enable bip2ml to reason about a considerably larger set of programs. bip2ml could be utilized to establish relational properties between code that makes use of **dynamic programming** techniques and its functional, simpler and slower counterpart. Finally, when we mentioned **program evolution**, we are referring to code versioning, so that refactoring or new features do not modify the behavior of code that we previously proved correct.

Although we are able to introduce some invariants automatically (thanks to the WhyRel rules), there is still a long way to go when we discuss the automation of the proof of program equivalence. We wonder if we could infer trivial variants and some other invariants, adding then those to specification attached to the final generated code. That would be a promising way of reducing weight off the shoulders of the users of bip2ml.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[2] A. J. Staff. *What happened to the fuel-control switches on doomed Air India flight 171?* Accessed on September 26, 2025. 2025. URL: https://www.aljazeera.com/news/2025/7/17/what-happened-to-the-fuel-control-switches-on-doomed-air-india-flight-171 (cit. on p. 1).

[3] N. G. Leveson. " The Therac-25: 30 Years Later ". In: *Computer* 50.11 (2017-11), pp. 8–11. ISSN: 1558-0814. DOI: 10.1109/MC.2017.4041349. URL: https://doi.ieeecomputersociety.org/10.1109/MC.2017.4041349 (cit. on p. 1).

[4] D. N. Arnold. *ARIANE 5 - Flight 501 Failure - Report by the Inquiry Board*. Accessed on September 26, 2025. 1996. URL: https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html (cit. on p. 1).

[5] U. G. A. Office. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. Accessed on September 26, 2025. 1992. URL: https://www.gao.gov/products/imtec-92-26 (cit. on p. 1).

[6] 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. 1).

[7] 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, 4).

[8] R. Nagasamudram, A. Banerjee, and D. A. Naumann. "WhyRel: an auto-active relational verifier". In: *International Journal on Software Tools for Technology Transfer* (2025), pp. 1–15 (cit. on pp. 2, 28).

[9] *OCaml's Reference Manual*. https://ocaml.org/manual/5.3/index.html. Accessed on February 6, 2025 (cit. on pp. 2, 9).

[10] 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 pp. 2, 22).

[11] A. Charguéraud et al. "GOSPEL - Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by M. H. ter Beek, A. McIver, and J. N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. DOI: 10.1007/978-3-030-30942-8_29. URL: https://doi.org/10.1007/978-3-030-30942-8%5C_29 (cit. on pp. 2, 13).

[12] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_31 (cit. on pp. 2, 14).

[13] 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 pp. 2, 11).

[14] 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. 4).

[15] 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. 5).

[16] 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. 5, 22).

[17] G. Barthe, J. M. Crespo, and C. Kunz. "Product programs and relational program logics". In: *J. Log. Algebraic Methods Program.* 85.5 (2016), pp. 847–859. DOI: 10.1016/J.JLAMP.2016.05.004. URL: https://doi.org/10.1016/j.jlamp.2016.05.004 (cit. on pp. 7, 9, 22).

[18] 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. 9).

[19] 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. 9).

[20] Anonymous. *Who Uses OCaml?* Accessed on February 6, 2025. n.d. URL: https://ocaml.org/industrial-users (cit. on p. 10).

[21] Anonymous. *An SMT Solver for Software Verification*. Accessed on February 6, 2025. n.d. URL: https://alt-ergo.ocamlpro.com/ (cit. on p. 10).

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

[23] 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. 10).

[24] 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. 11).

[25] 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. 17, 22).

[26] 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. 19).

[27] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://www.worldcat.org/oclc/01958445 (cit. on p. 19).

[28] 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. 19).

[29] 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. 19).

[30] 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. 22).

[31] 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. 22).

[32] G. Barthe et al. "Verifying relational properties using trace logic". In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2019, pp. 170–178 (cit. on p. 28).

[33] A. Banerjee et al. "A Relational Program Logic with Data Abstraction and Dynamic Framing". In: *ACM Trans. Program. Lang. Syst.* 44.4 (2023-01). ISSN: 0164-0925. DOI: 10.1145/3551497. URL: https://doi.org/10.1145/3551497 (cit. on p. 29).

[34] B. R. Churchill et al. "Semantic program alignment for equivalence checking". 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. 1027–1040. DOI: 10.1145/3314221.3314596. URL: https://doi.org/10.1145/3314221.3314596 (cit. on p. 30).

[35] Anonymous. *ocamllex produces a lexical analyzer from a set of regular expressions with associated semantic actions*. Accessed on September 16, 2025. n.d. URL: https://ocaml.org/manual/5.3/lexyacc.html (cit. on p. 39).

[36] F. Pottier and Y. Régis-Gianas. *Menhir is a LR(1) parser generator for the OCaml programming language. That is, Menhir compiles LR(1) grammar specifications down to OCaml code.* Accessed on September 16, 2025. n.d. URL: https://gallium.inria.fr/~fpottier/menhir/ (cit. on p. 39).

[37] L. Jane Street Group. *Dune is a build system for OCaml projects. Using it, you can build executables, libraries, run tests, and much more.* Accessed on September 22, 2025. 2018. URL: https://dune.build/ (cit. on p. 39).

[38] O. Labs. *VSCode OCaml Platform is a Visual Studio Code extension for OCaml and relevant tools.* Accessed on September 16, 2025. n.d. URL: https://github.com/ocamllabs/vscode-ocaml-platform (cit. on p. 39).

I

Annex 1 - Conditionally Aligned Loops

BipLang + GOSPEL

```
(*@ axiom mult: ∀ a:int, b:int, c:int, d:int.
   0 < a → 0 < b → 0 < c → 0 < d → a > b → c > d →
     (a * c) > (b * d) *)

let cond_align_loops_biplang (|_x : int_|, |_n : int_|) : |_int_| = begin
  let y = |_ ref x _| in
  let z = ref 24 <|> ref 16 in
  let w = |_ ref 0 _| in

  while !y > 4 <|> !y > 4 . (!w mod n ≠ 0) <|> (!w mod n ≠ 0) do
    (*@ invariant !y_l = !y_r && !y_l ≥ 4
        invariant !z_r > 0 && !z_l > !z_r *)

    if (((!w mod n) = 0) <|> ((!w mod n) = 0)) then begin
      z := (!z * !y <|> !z * 2);
      y := |_ !y - 1 _|
    end else begin () end;

    w := |_ !w + 1 _|
  done;

  |_ !z _|
end
(*@ requires x_l = x_r && x_l ≥ 4 && n_l = n_r && n_l > 0
    diverges
    ensures  match result with (l_res, r_res) → l_res > r_res *)
```

Figure I.1: Conditionally aligned loops (BipLang).

OCaml + GOSPEL

```
(*@ axiom mult: ∀ a:int, b:int, c:int, d:int.
   0 < a → 0 < b → 0 < c → 0 < d → a > b → c > d →
     (a * c) > (b * d) *)

let cond_align_loops_ocaml (x_l : int) (x_r : int)
  (n_l : int) (n_r : int) : int * int =

  let y_l = ref (x_l) in
  let y_r = ref (x_r) in
  let z_l = ref (24) in
  let z_r = ref (16) in
  let w_l = ref (0) in
  let w_r = ref (0) in

  while ((!y_l > 4) || (!y_r > 4)) do
    (*@ invariant !y_l = !y_r && !y_l ≥ 4
        invariant !z_r > 0 && !z_l > !z_r
        invariant (!y_l > 4 && mod !w_l n_l ≠ 0) ||
                  (!y_r > 4 && mod !w_r n_r ≠ 0) ||
                  (¬ (!y_l > 4) && ¬ (!y_r > 4)) ||
                  (!y_l > 4 && !y_r > 4) *)

    if ((!y_l > 4) && ((!w_l mod n_l) ≠ 0))
    then begin
      if ((!w_l mod n_l) = 0)
      then begin
        z_l := (!z_l * !y_l);
        y_l := (!y_l - 1)
      end else begin
        ()
      end;
      w_l := (!w_l + 1)
    end else begin
      if ((!y_r > 4) && ((!w_r mod n_r) ≠ 0))
      then begin
        if ((!w_r mod n_r) = 0)
        then begin
```

```
          z_r := (!z_r * 2);
          y_r := (!y_r - 1)
        end else begin
          ()
        end;
        w_r := (!w_r + 1)
      end else begin
        assert ( (((!w_l mod n_l) = 0)) = (((!w_r mod n_r) = 0)) );
        if ((!w_l mod n_l) = 0)
        then begin
          z_l := (!z_l * !y_l);
          z_r := (!z_r * 2);
          y_l := (!y_l - 1);
          y_r := (!y_r - 1)
        end else begin
          ()
        end;
        w_l := (!w_l + 1);
        w_r := (!w_r + 1)
      end
    end
  done;

  (!z_l, !z_r)
(*@ requires x_l = x_r && x_l ≥ 4 && n_l = n_r && n_l > 0
    diverges
    ensures  match result with (l_res, r_res) → l_res > r_res *)
```

Figure I.2: Conditionally aligned loops (OCaml).