Pedro Miguel Laforêt Barroso

Bachelor Degree

Formally Verified Bug-free Implementations of (Logical) Algorithms

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: António Maria Lobo César Alarcão Ravara,
Associate Professor,
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-adviser: Mário José Parreira Pereira,
Post-Doctoral Researcher,
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Examination Committee

Chairperson: José Júlio Alves Alferes
Raporteur: Nelma Resende Araújo Moreira
Member: António Maria Lobo César Alarcão Ravara

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2019

Formally Verified Bug-free Implementations of (Logical) Algorithms

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Para as mulheres incríveis da minha vida: mãe, avó e Joana.

Acknowledgements

I would like to start to express my special thanks of gratitude to my advisor Professor António Ravara and co-advisor Mário Pereira for the continuous support, availability, motivation and immense knowledge. Their guidance was fundamental, I could not imagine having finished this dissertation without them.

To the Tezos Foundation that kindly supported this dissertation.

To the Faculty of Sciences and Technology of New University of Lisbon and in special to the Department of Informatics for all the conditions and well hours spent, I felt like it was my second home.

To the P3/14 colleagues for receiving me so well and all the positive vibes.

To my college group, Diogo Carrasco, Nuno Madaleno, João Neves, Tiago Conceição, João Borralho and João Pacheco, who has always accompanied me from day one and helped make this adventure in Computer Science even more fruitful.

To Ricardo Alves, for changing my mood and showing me that work is not everything, that I also need to breathe.

To my girlfriend, Joana Cristóvão, for always believing in me, for showing me that I was capable, and for the patience to put up with me even when I don't put up with myself.

Last but not least, I would like to thank my mom and grandmom for their wise counsel and sympathetic ear. You are always there for me.

If you never try you'll never know.

Abstract

Notwithstanding the advancements of formal methods, which already permit their adoption in a industrial context (consider, for instance, the notorious examples of Airbus, Amazon Web-Services, Facebook, or Intel), there is still no widespread endorsement. Namely, in the Portuguese case, it is seldom the case companies use them consistently, systematically, or both. One possible reason is the still low emphasis placed by academic institutions on formal methods (broadly consider as developments methodologies, verification, and tests), making their use a challenge for the current practitioners.

Formal methods build on logics, "the calculus of Computer Science". Computational Logic is thus an essential field of Computer Science. Courses on this subject are usually either too informal (only providing pseudo-code specifications) or too formal (only presenting rigorous mathematical definitions) when describing algorithms. In either case, there is an emphasis on paper-and-pencil definitions and proofs rather than on computational approaches. It is scarcely the case where these courses provide executable code, even if the pedagogical advantages of using tools is well know.

In this dissertation, we present an approach to develop formally verified implementations of classical Computational Logic algorithms. We choose the Why3 platform as it allows one to implement functions with very similar characteristics to the mathematical definitions, as well as it concedes a high degree of automation in the verification process. As proofs of concept, we implement and show correct the conversion algorithms from propositional formulae to conjunctive normal form and from this form to Horn clauses.

Keywords: Formal Methods; Computational Logic; Propositional Algorithms; Program verification; Functional language; Why3

Resumo

Não obstante os avanços dos métodos formais, que já permitem sua adoção num contexto industrial (considere, por exemplo, os conhecidos casos da Airbus, Amazon Web-Services, Facebook ou Intel), ainda não existe uma ampla adoção. Nomeadamente, no caso português, raramente as empresas os usam de maneira consistente ou sistemática. Uma possível razão é a baixa ênfase colocada pelas instituições acadêmicas em métodos formais (geralmente considerados como metodologias de desenvolvimento, verificação e testes), o que torna o seu uso num desafio para os praticantes atuais.

Os métodos formais baseiam-se na lógica, "a lingua franca da ciência da computação". A Lógica Computacional é, portanto, um campo essencial da Ciência da Computação. Os cursos nesta matéria geralmente são muito informais (apenas fornecendo especificações em pseudo-código) ou muito formais (apenas apresentando definições matemáticas exigentes) ao descrever algoritmos. Nos dois casos, existe ênfase em definições e provas em papel e lápis, e não em abordagens computacionais. É raro o caso em que os cursos fornecem código executável, mesmo que as vantagens pedagógicas do uso de ferramentas sejam conhecidas.

Nesta dissertação, apresentamos uma abordagem para desenvolver implementações formalmente verificadas de algoritmos clássicos de lógica computacional. Escolhemos a plataforma Why3, pois permite implementar funções com características muito semelhantes às definições matemáticas, além de conceder um alto grau de automação no processo de verificação. Como provas de conceito, implementamos e mostramos corretos os algoritmos de conversão de fórmulas proposicionais para a forma normal conjuntiva e desta para cláusulas de Horn.

Palavras-chave: Lógica Computacional; Algoritmos de lógica proposicional; Verificação de programas; Linguagem funcional; Why3

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents a brief introduction to the work developed in this dissertation. Starting with the contextualization (Section 1.1), a presentation of the FACTOR project (Section 1.2), the identification of the problem (Section 1.3), the definition of the objective (Section 1.4) and the list of contributions (Section 1.5). Lastly, it is described the structure of the dissertation (Section 1.6).

1.1 Context

Software bugs and vulnerabilities is a common concern, not only for computer scientists but also for the general public. News about the discovery of bugs and vulnerabilities are published almost every day, which is making the societal tolerance to these situations to decrease rapidly. Although some bugs are insignificant, having no consequences, there are others that are catastrophic and unacceptable, implying severe financial consequences or, even more seriously, a threat to human well-being.

For example, the famous Ariane 5 Flight 501 bug was that software tried to fit a 64-bit number into a 16-bit space causing the crash of both primary and backup computers [42], forcing engineers to push the self destruct button and, according to the Media, lose approximately half billion dollars.

There is also recent estimations that 1,000 deaths per year are caused in the English NHS by unnecessary bugs in their software [69].

The ambition to develop completely bug-free programs has existed nearly as long as the field of computer science. However, verifying every single program execution scenario is a challenge, as it is tough to imagine every single scenario and every single behaviour of our programs. Nonetheless, programmers still make assumptions about the correctness of their programs based on tests.

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Edsger Dijkstra

Program verification ensures that programs, under some defined specification, are correct. So, to ensure that a program is free of bugs, we should turn ourselves to it. However, most developers have only habits for testing and almost none for program verification. They do not use methodologies or tools for verification on a daily basis of programming. We shall later present a more in-depth overview of program verification.

1.2 FACTOR

This dissertation is part of the FACTOR (Functional ApproaCh Teaching pOrtuguese couRses) project[1]. FACTOR is funded by the Tezos Foundation [67] and aims to promote the use of OCaml [40] in the Portuguese academic community, namely through the support of teaching approaches and tools. In particular, it aims to broaden and consolidate the user base of software and teaching materials in OCaml in the Portuguese community for subjects in the area of Computational Logic and Fundamentals of Computing in Computer Science courses.

The FACTOR project makes a modest contribution in two fundamental aspects: implementation and verification of classical algorithms of Computational Logic and web support for students to run these implementations trough step-by-step executions.

1.3 Problem

Every ten years, there is a world congress on formal methods; the last one was in Porto, in October 2019. In this congress, there was a uniform opinion that nowadays there is a mature approach to formal methods. However, it is still far from being general, even if the number and quality of tools is impressive.

Foundational courses in Computer Science, like Computational Logic, aim at presenting key basilar subjects to the education of undergraduate students. To strength the relation of the topics covered to sound programming practices, it is relevant to link the mathematical content to clear, executable and correct implementations.

In some cases, these courses could have a more computational approach. When too informal, only pseudo-codes are provided, which hampers the understanding of all the aspects of the algorithm, as they are far from the definitions and not executable. On the other hand, when formal, courses decide to describe algorithms close to their mathematical properties and proofs, for instance in set theories, making it hard to handle.

[1] https://releaselab.gitlab.io/factor/

Implementations of algorithms and tools that help students go through the resolutions of exercises are crucial, as they are an important pedagogical object with proved results [55].

Another aspect to mention is the lack of material about correction. As mentioned in Section 1.1, program verification is an important aspect of software development, since it allows the development of bug-free software. However, some courses make only a small effort on showing the presented algorithms correct even though this can also provide a deeper knowledge of them.

In the Portuguese situation, it seems that formal methods are gradually being inserted. However, from what can be assessed by internships, theses, and work in the Integrated Masters in Computer Science, the majority develops projects without recourse to the formal methods. For instance, the same course of the Faculty of Science and Technology of NOVA University of Lisbon only starts referring to program verification in Software Construction and Verification. A course for 4th-year students that is not mandatory.

1.4 Objective

The main objective of this dissertation is to contribute to the development of pedagogical material to support the Computational Logic courses, through the implementation and verification of standard and classical algorithms of Computational Logic.

The objective is to give **supplementary** pedagogical material to a course (mostly) taught in the second year of Computer Science. The implementations should be easily related to the algorithms described in the lectures, specially to their mathematical definitions. Additionally, these implementations should support step-by-step executions, which adds the ability to slowly follow each step of the algorithm, resulting in a better algorithm apprenticeship. This will be achieved using CPS (Continuation-passing style) or Defunctionalization transformations.

Functional languages are specially suitable to handle symbolic computation and are based on functions. This type of languages has a closer relationship to the mathematical definitions, making them suitable for undergraduate students.

Once an algorithm is implemented we need to verify it. This needs to be as much automated as possible, since the students in the second year of Computer Science have not, or merely, acquired knowledge about program verification. Taking this into consideration and also that the implementations are over propositional logic, or at the maximum first-order logic, the Why3 program verification platform fits perfectly. Why3 will be later described in Section 2.3.

1.5 Contributions

The main objective of this dissertation, as mentioned in the previous section, is to develop pedagogical material to support Computational Logic courses. We contribute with clear identification of the properties and specifications of two algorithms, one that converts propositional formulae into Conjunctive Normal Form (CNF) and the other from CNF to Horn clauses. The identification will help the relation between the implementation and the mathematical definitions, and also support the correspondent correctness criteria and proof.

Given the importance of step-by-step executions, we also provide CPS/Defunctionalization transformations. In the future, these will be used and incorporated in a web environment that will help and guide students throughout the algorithms.

In summary, the contributions are:

1. Clear identification of properties and specifications of two propositional logic algorithms.
2. Presentation of their correspondent correctness criteria.
3. Definition of correctness proofs of their implementations using Why3.
4. CPS and Defunctionalised versions of the CNF conversion algorithm, also proved corrected.

1.6 Document Structure

The remainder of this dissertation is organised as follows:

- Chapter 2 - State of the Art reports the aspects of Computational Logic courses, the existing tools to support them and the main proof-assistant tools.
- Chapter 3 - Background does in-depth view about program verification, continuation-passing style and defunctionalization; it also presents a simple proof in Why3.
- Chapter 4 - Supporting Boolean Theories defines the theories used in our implementations.
- Chapter 5 - Transformation Algorithm to Conjunctive Normal Form presents the implementation and verification of the algorithm that converts propositional formulae to CNF.
- Chapter 6 - Transformation Algorithm from CNF to Horn Clauses presents the implementation and verification of the algorithm that converts CNF formulae to Horn Clauses.
- Chapter 7 - Towards Step-by-Step Execution shows the CPS and Defunctionalization transformation of the implementations.

- Chapter 8 - Conclusions synthesizes the work developed and presents the conclusions taken from the study conducted in this dissertation.

Chapter 2

State of the Art

In Chapter 1 we made an introduction to the context and problem of this dissertation, mentioning the importance of Computational Logic courses, the lack of tools and implementations of algorithms, and the little emphasis about program verification.

This chapter presents a global overview of Computational Logic courses in Portugal, contrasting with the ACM and IEEE recommendations, and the actual tools to support these courses. Still in this chapter we present and compare the most common proof-assistant tools.

2.1 Current State of Computational Logic in Portugal

According to the 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science of ACM and IEEE [23] basic logic (propositional and first-order logic), proof techniques and formal methods should be covered, respectively, in 9, 10 and 4.5 core hours.

Computational Logic is an essential field of Computer Science. Symbolic systems are at the roots of Informatics, with implications in many areas. The development of appropriate solutions to problems requires rigorous approaches, based on precise formal models that ensure the quality and correctness of the systems built. It is a mandatory unit usually taught in the second year of the Computer Science course that covers the principal aspects of propositional and first-order logic. This unit is very important for Computer Science students, as they should learn to reason about computer programs in a more mathematical way.

In this section we present the global overview of the current state of Computational Logic in Portugal. The overview here described is based on "Report on the Curricular

Unit Computational Logic" of the professor António Ravara [61] and in the information available online (December 2019).

FCT-NOVA

The Integrated Master in Computer Science has the course Computational Logic, that is a compulsory curricular unit, being taught in the first semester of the second year. Its weekly workload is 2 theoretical hours and 3 practical hours, corresponding to 6 ECTS credit units. In the unit, an introduction to the first order logic, focused on the notions of formal language and argumentation as well as its formalization in deduction systems, is studied. Two systems (natural deduction and resolution) are studied. Emphasis is placed on the practical use of logic for problem-solving, that is, on its computational aspects [45]. The bibliographic's main reference is "Language Proof and Logic (2nd edition)" [9].

University of Beira Interior

The degree in Computer Science and Engineering offers a Computational Logic course, a compulsory curricular unit thaught in the first semester of the second year. The references are "Logic in Computer Science: Modelling and Reasoning about Systems" [35] and "Rigorous Software Development, An Introduction to Program Verification" [5]. The course has also important additions: all the algorithms presented in the lectures are implemented in OCaml and the syllabus includes Boolean Satisfiability problem solving (SAT) and Satisfiability Modulo Theories (SMT) [46]

IST

The degree in Informatics Engineering and Computers has the course Logic for Programming [43] that teaches propositional and first-order logic, resolution, and Prolog. It is a compulsory curricular unit thaught in the second semester of the first year. The bibliographic references are the following: "Lógica e Raciocínio" [48], "Mathematical Logic for Computer Science" [13]; and "Logic in Computer Science: Modelling and Reasoning about Systems" [35].

University of Algarve

The course Logic and Computation [37] addresses propositional and first-order logic (syntax, semantics, and deductive systems). The bibliographic references are "Mathematical Logic for Computer Science" [13] and "Introduction to the Theory of Computation" [64].

University of Aveiro

The degree in Informatics Engineering does not addresses a specific course on logic. Propositional and first-order logics are part of the Discrete Mathematics course [50]. The bibliographic main references are: "Matemática Discreta" [19]; "Discrete Mathematics" [14]; "Concrete Mathematics - A Foundation for Computer Science" [30]; and "Applied and Algorithmic Graph theory" [20].

University of Minho
The course Lógica EI [47] addresses propositional and first-order logic (syntax, semantics, and deductive systems). The bibliographic main reference is "Logic and Structure" [24].

University of Lisbon
The degree in Informatics Engineering has a course on First-Order Logic. The main bibliography reference is again "Language, Proof and Logic" [9].

Faculty of Engineering of University of Porto
The degree in Informatics Engineering does not addresses a specific course on logic. Propositional and first-order logics are part of the Discrete Mathematics course [49]. The bibliographic main references are "Language, Proof and Logic" [9] and "Discrete Mathematics with Graph Theory" [29].

Faculty of Sciences of University of Porto
The degree in Computer Science and its integrated master's degree in Network and Information Systems Engineering have a course on Computational Logic in the first semester of the second year [44]. It covers Propositional and First-Order Logic (syntax, semantics, algorithms for satisfiability checking, deductive systems, and resolution) and automatic demonstration in prolog of Horn clauses, unification and resolution. In the exercises of the course, students need to implement specific algorithms in a language of their choice. The main bibliography reference is again Huth and Ryan, and the Lecture Notes of Nelma Moreira [52].

Conclusion. The Logic courses in the Computer Science degrees in Portugal mostly follow the ACM and IEEE recommendations. The courses including a computational view, present satisfiability algorithms and the resolution method. Barwise & Etchemendy and Huth & Ryan are the most common references [61].

Only the courses at UBI provides to the students implementations of all the basic algorithms in a functional language (OCaml). Some universities provides implementations closer to Prolog. However, in contrast to OCaml, Prolog does not have a strong type discipline. Every data is treated as a unique type, which nature depends of the way its have been declared. This is a huge disadvantage noticeably when defining grammars as types.

On the other hand, if we search for verification of Prolog programs, few results emerge. The main works seems to be PrologCheck, an automatic tool for property based testing of Prolog programs, that given a specification it randomly generates test cases for the properties to be tested, executing them and estimating their validity. It is inspired by the QuickCheck tool, that was originally designed for Haskell [6]. It would be interest to use the tool to achieve our goals and compare the results.

The "Formal Methods: A First Introduction using Prolog to specify Programming Language Semantics" [31] presents the idea of using first-order Horn Clauses for specification and proof scripting language, contributing also, with a module that facilitates

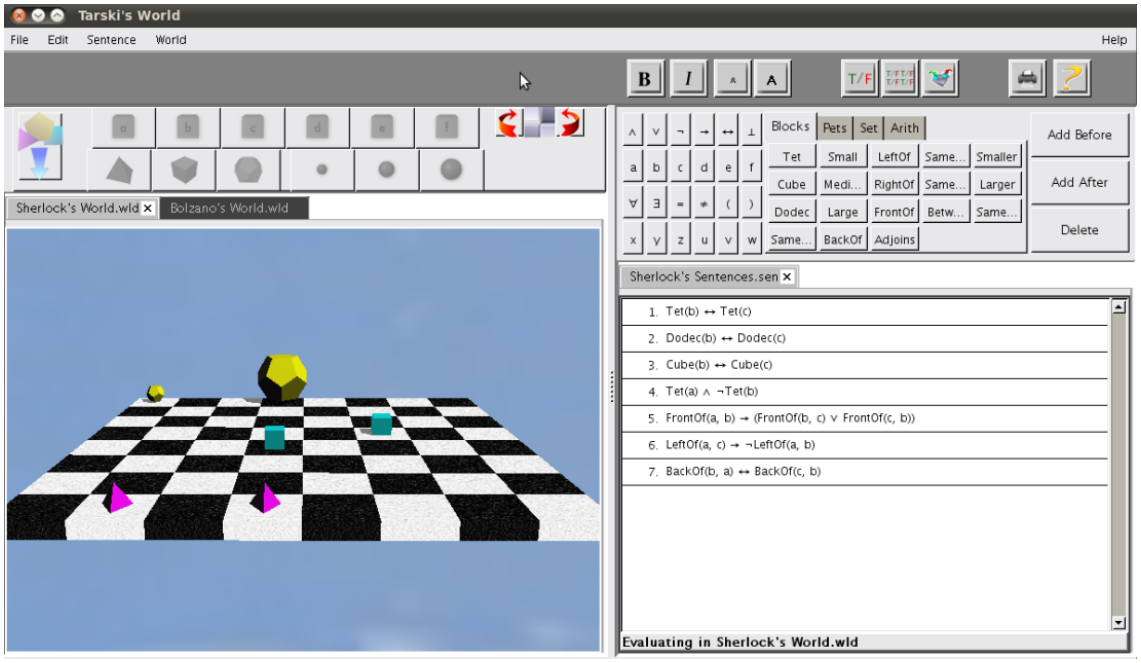

Figure 2.1: Tarski's World main window.

the use of Prolog as a proof assistant. Again, it would be interesting to see if our aims are feasible in this setting.

Finally, there is an analyser for verifying the correctness of a Prolog program [22]. It combines, adapts and sometimes improves various existing static analyses in order to verity total correctness of Prolog programs regarding to formal specification.

2.2 Tools to support Computational Logic

Support tools have an important role in the process of learning something, it helps to understand the concept of the given matter, while it also increases the motivation of the student [55]. In this section we present the actual existing tools to support Computational Logic.

2.2.1 Tarski's World

Tarski's World is an essential tool for helping students learning the language of logic. The program allows students to build three-dimensional worlds and then describe them in first-order logic [8]. The program is more like a game where students can create worlds and make sentences about the constructed world, the program then evaluates the sentences and give feedback to the user.

Figure 2.1 shows the interface, which is very intuitive and is easy to use. Students can easily play with the program and, at the same time, learn.

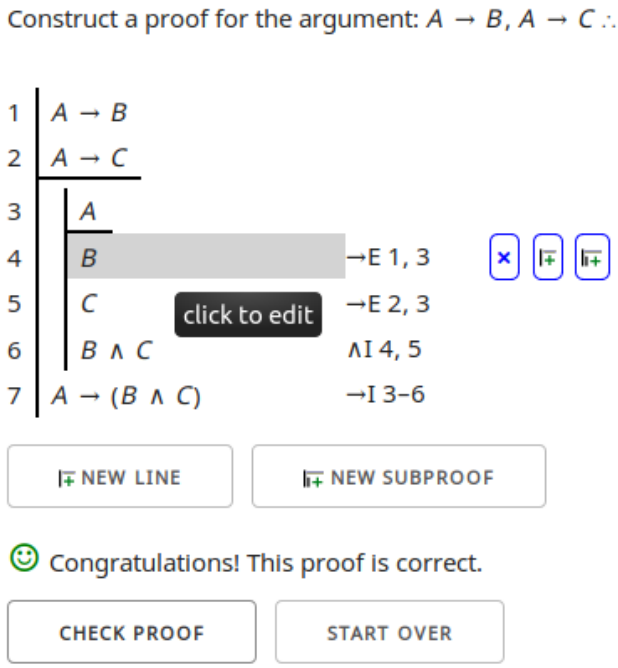

Figure 2.2: Fitch-checker $A \to B, A \to C \therefore A \to (B \wedge C)$ proof.

2.2.2 Fitch

Fitch notation is a system for constructing natural deduction proofs used in propositional and in first-order logic. Fitch-style proofs arrange the sequence of sentences that make up the proof into rows [1]. It has a unique feature where the degree of indentation of each row indicates which assumptions are active for that step.

Here is an example that prove that P ↔ ¬¬ P: *(from Fitch Notation Wikipedia page)*:

```
1  |__                          [assumption, want P iff not not P]
2  |   |__ P                    [assumption, want not not P]
3  |   |   |__ not P            [assumption, for reductio]
4  |   |   |   contradiction    [contradiction introduction: 2, 3]
5  |   |   not not P            [negation introduction: 3]
6  |
7  |   |__ not not P            [assumption, want P]
8  |   |   P                    [negation elimination: 7]
9  |
10 |   P iff not not P          [biconditional introduction: 2 - 5, 7 - 8]
```

Fitch-checker helps the construction and validation of this style of deduction. Figure 2.2 shows the proof of $A \to B, A \to C \therefore A \to (B \wedge C)$ with this tool.

2.2.3 Boole

Boole [17] is a simple tool that helps students to build and verify truth tables. The figure 2.3 shows the user interface of the Boole program.

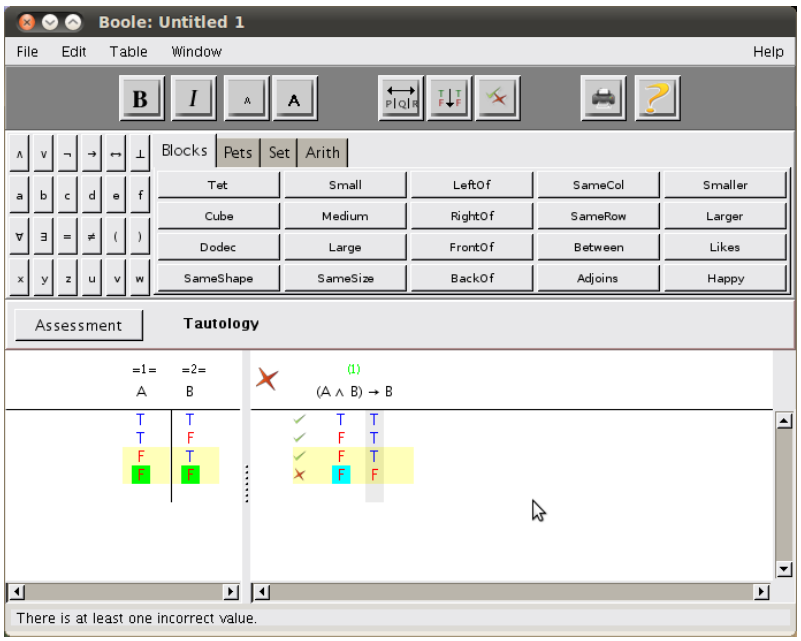

Figure 2.3: Boole program.

2.3 Proof-Assistant Tools

These interactive tools are computer systems to help the development of formal proofs by human-machine collaboration [27]. In this section, we present the most popular tools based on a functional programming language.

2.3.1 Agda

Agda [18] is a dependently typed functional programming language. It is based on intuitionistic type theory and has many similarities with other proof assistants based on dependent types, such as Coq. However, in constrast to Coq it has no support for tactics [1]. The user sees a partially proof term during the process. The incomplete parts of the term are represented by place-holders, called `meta variable` [41].

2.3.2 Coq

Coq [10] is an interactive proof assistant based on type theory. Coq uses the Calculus of Inductive Constructions that itself combines both a higher-order logic and a richly-typed functional programming language. Similar to Why3, Coq can extract certified OCaml programs. In proofs, Coq combines two languages: Gallina and Ltac. The statement for proof and structures it relies on are written in Gallina, a functional language, while the proof process is being treated by the commands written in Ltac, a procedural language for manipulating the proof process [68].

[1] A tactic replaces the goal with the sub-goals it generates.

2.3.3 Isabelle/HOL

Isabelle/HOL [56] is a simply typed higher-order logic inside the logical framework Isabelle. It is not primarily intended as an platform for program verification and does not contain specific syntax for stating *pre* and *post* conditions. Similar to the previous theorem provers described, Isabelle/HOL also has his special ML programming language. It combines two languages: HOL as a functional language and Isar as a language for describing procedures to manipulate the proof [21].

2.3.4 Twelf

Twelf [59] provides meta-language used to specify, implement, and prove properties of deductive systems such as programming languages and logics. It relies on the LF type theory developed by Frank Pfenning and Carsten Schürmann [58]. Twelf can be used as a logic programming language. However, there is no sophisticated operators (such as ones for performing I/O), which turns it less well-suited for practical logic programming applications. The Twelf theorem proving component is at an experimental stage (2016) [71, 72].

2.3.5 Why3

Why3 is a platform for deductive program verification. The Why3 objective is to provide as much automation as possible when performing proofs, which distinguishes it from other platforms. It provides a language for specification and programming, called WhyML (a first-order language with polymorphic types, pattern matching, and inductive predicates), a mechanism to extract certified OCaml programs and support to third-party theorem provers, both automated and interactive. This is an added value when the user wants to use different provers, rather than stick to just one. Why3 standard library is formed of many logic theories (in particular for integer and floating point arithmetic, sets, and dictionaries) and basic programming data structures. It also has support for partial functions [16].

2.3.6 Advantages and Disadvantages

Proof-Assistant Tool	Language Order	User Effort	Automation
Agda	Higher-Order	Medium	None
Coq	Higher-Order	Medium	Low
Isabelle/HOL	Higher-Order	Medium	Medium
Twelf	Higher-Order	Medium	Low
Why3	First-Order	Low	High

Table 2.1: Comparison of Proof-Assistant Tools.

When choosing a proof-assistant tool it is important to relate each one's capabilities according to our needs. Many factors may suggest the use of one proof-assistant over the others. In the Table 2.1, some of these factors are described, i.e, language order, user effort and automation. Comparisons between them are, however, scarce [73, 74].

All the described proof-assistant support first and higher-order languages (except Why3 that focus on first-order, providing only higher-order "curried" syntax). The user effort point of view is related to the degree of automation, as the more automation, the less effort the user will have.

Why3 is suited for proofs that allow for more automation and to properties that are mainly focus in first-order language, as Coq and Isabelle require more expertise but offers support to higher-order. Agda power as a programming language is similar to Coq (with main differences being pattern matching). However, does not support tactics, notations, extraction and wide library, which hampers its use effort.

Twelf differs from the rest, as the proofs are typically developed by hand and does not have the full potential as a programming language.

2.3.7 Conclusions

This dissertation aims at contributing to increase the use of OCaml in Portuguese Universities, proving that they are suitable for algorithm presentations and simple proofs. The majority of the proof-assistant tools presented offer some ML-like language. However, only Coq and Why3 can extract certified OCaml programs. The difference between Coq and Why3 is that the former is based on a first-order logic with inductive predicates and automatic provers, and Coq on an expressive theory of higher-order logic [21]. Crucially for this work, Why3 provides a degree of automation suitable for undergraduate students, where Coq is interactive and require much more expertise.

Chapter 3

Background

Chapter 2 presents some proof-assistant tools, and concludes that Why3 is the ideal tool for the implementation and verification of classical logic algorithms. This chapter presents the concept of program verification, a simple proof using Why3 (we use a factorial function as an example). Lastly, we discuss the continuation-passing style and defunctionalization as a way to have an explicit stack structure. Therefore, in the future, it will be possible to introduce a mechanism that allows step-by-step executions.

3.1 Program Verification

I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic.

C. A. R. Hoare

Program verification is the use of formal, mathematical techniques to ensure that a program is correct. The process begins with the formal description of a specification for a program in some symbolic logic, following with a proof that the program meets the formal specification. The analogy to a sequent in program verification is a Hoare triple [33], so named because it is made up of three components: precondition (B), program (P) and postcondition (A).

$$\{B\}\ P\ \{A\}$$

The triple means that, if the precondition B is true in a state before the execution of the program P, then the resulting state will satisfy the postconditions A.

Partial versus Total Correctness. The Hoare triple mentioned before is a partial correctness triple for specifying and reasoning about the behaviour of a program. A partial correctness triple because assuming that the precondition is true just before the program executes, then `if the program terminates`, the postcondition is true. In order to be totally correct, program's termination must be guaranteed. Therefore, total correctness is partial correctness plus termination.

Weakest Precondition. Consider the correct Hoare triple {y = 2} y = y * 2 {y > 0}. Although correct, this Hoare triple is not as precise as it could be. Namely, we could have a stronger postcondition. For example, `y < 10 && y > 2` is stronger (gives more information). The strongest and most useful postcondition we could have is y = 4. So, if {B} P {A} and for all A' such that {B} P {A'}, the A' $\implies$ A, then we are facing the strongest postcondition of S with respect to the B. The same way in the opposite direction. If {B} P {A} and for all A' such that {A'} P {B}, B' $\implies$ B, then we are facing the weakest precondition `wp(P,A)` of P with respect to the A [4].

Edsger Dijkstra had a major role in the automation of deductive program verification with his work for weakest precondition calculus [25]. The idea behind is that given a statement P, the weakest-precondition of P is a function, denoted `wp(P,postcondition)`, that computes the weakest precondition on the initial state ensuring that execution of P terminates in a final state satisfying the postcondition. Therefore, we can write the following valid Hoare triple:

$$\{wp(P,A)\}\ P\ \{A\}$$

The validity of a Hoare triple {B} P {A} is provable in Hoare logic for total correctness if and only if the first-order predicate below holds:

$$B \implies wp(P,A)$$

The weakest precondition calculus is the core of some automatic verification condition generators (VCG). Why3, the verification tool used as the platform for deductive program verification in this dissertation, is one of these tools. There are however, other tools like Dafny [39], VeriFast [38] and Viper [54], based on separation logic instead of first-order logic.

The VCG takes as input a program along with the desired specification and generates a set of logical statements called verification conditions. Then the second part of the verification process starts, prove the validity of these logical statements. For that, one uses theorem provers, a standard approach to discharge verification conditions, or if one wants more automation uses SMT's (Satisfiability Modulo Theory) solvers [11], for example, Alt-Ergo [15], CVC4 [12] and Z3 [53].

3.2 Why3

A brief description of Why3 was presented in Section 2.3.5. The purpose of this section is to see Why3 in action: we will present the implementation and verification of a simple example.

Considering the following naive factorial function implemented in WhyML:

```
let rec fact_naive (x: int) : int
  = if x = 0 then 1
    else x * fact_naive (x - 1)
```

The first step is to reason about the properties of the program. The factorial function needs to receive as input a positive integer, otherwise, the program will enter in a loop and not terminate. This is called a precondition of the program and is indicated with the `requires` statement:

```
let rec fact_naive (x: int) : int
    requires{ x ≥ 0 }
  = ...
```

For the postcondition, the factorial function provided in the standard library of Why3 is used as specification, thus ensuring that the result of the function is equal to the result of the standard factorial function. The postconditions are indicated with the `ensures` statement:

```
let rec fact_naive (x: int) : int
    requires{ x ≥ 0 }
    ensures{ result = fact x }
  = ...
```

This specification ensures the partial correctness of the `fact_naive` function. To prove the total correctness, as seen in Section 3.1, we must ensure termination. In Why3 we prove termination using the `variant` statement. We must provide a variable of our program that in each iteration of the program is closer to 0. In our example, the variant is the variable x:

```
let rec fact_naive (x: int) : int
    requires{ x ≥ 0 }
    variant{ x }
    ensures{ result = fact x }
  = ...
```

The graphical interface of the proof can be observed in the Figure 3.1.

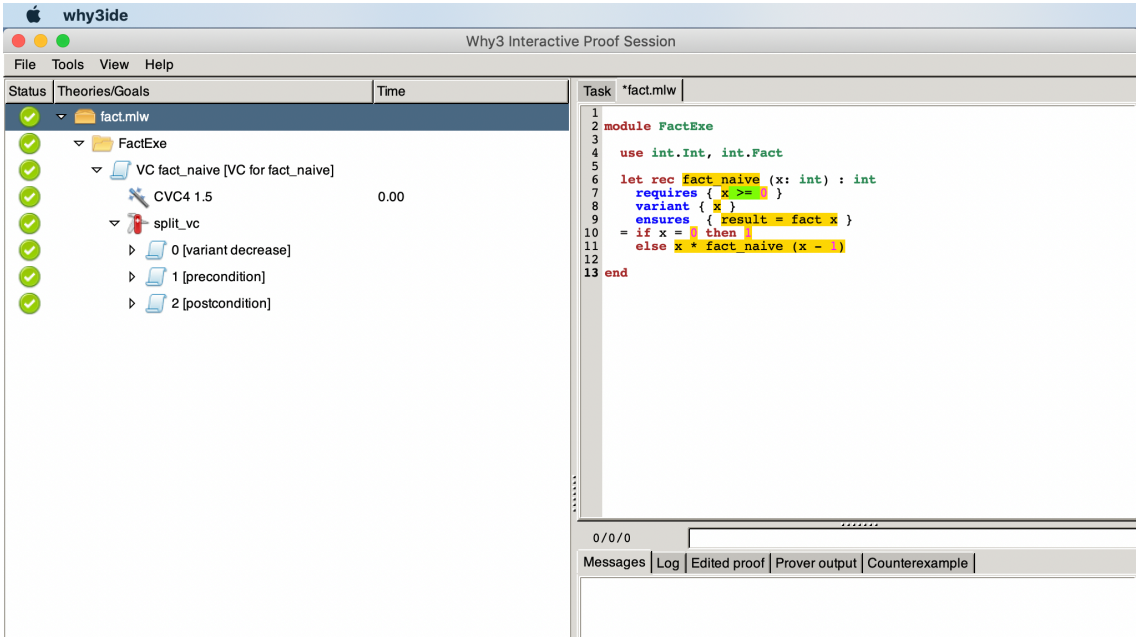

Figure 3.1: Proof of factorial function using Why3

3.3 Continuation-Passing Style

`Continuation-Passing Style` (CPS) is a programming style where the control is passed explicitly in the form of a continuation [7]. Normal functions take in some arguments, perform some computations and then return the result. CPS functions will have an extra parameter called a continuation, a function itself. The idea behind the CPS is to tell the program how to continue after getting the desired value (result), thus making the return an explicit action; instead of returning the result of the computation, the program calls the continuation [66].

Considering the naive factorial function written presented in the section above:

```
let rec fact (x: int) : int
  = if x = 0 then 1
    else x * fact (x - 1)
```

To transform the function into CPS, we need to add an extra parameter (the continuation) and use it instead of returning the values. Therefore, we add the continuation `k` to our factorial function:

```
let rec fact (x: int) (k: int → int) : int
...
```

Henceforth, we need to use this `k` function instead of returning the values. For `x = 0`, we just apply our `k` function to 1 (the returning element). For the remaining cases, when `x` is bigger than 0, we write a new continuation that combines the current element `x` with the result of the future continuations (`arg`), which is then passed to the next function iterations:

```
let rec fact_cps (x: int) (k: int → int) : int
  = if x = 0 then k 1
    else fact_cps (x - 1) (fun arg → k (x * arg))
```

In Table 3.1 is present the computation of the Factorial of 5. Practically, this computation can be seen as: (5 * (4 * (3 * (2 * (1 * 1))))).

Iteration #	**Result**
First Call	fact_cps (5) (fun arg -> arg)
1st Iteration	fact_cps (4) (fun arg5 -> arg (5 * arg5))
2nd Iteration	fact_cps (3) (fun arg4 -> arg5 (4 * arg4))
3rd Iteration	fact_cps (2) (fun arg3 -> arg4 (3 * arg3))
4th Iteration	fact_cps (1) (fun arg2 -> arg3 (2 * arg2))
5th Iteration	fact_cps (0) (fun arg1 -> arg2 (1 * arg1))
6th Iteration	arg1 1

Table 3.1: Computation of the Factorial of 5.

At each iteration, a continuation `k` is received, a new one is created and passed down to the next iteration. The continuation contains the deferred operations of the previous iterations [70].

The main advantage is the total control over program flow. In CPS there is no sequence of statements. Instead, each statement has an explicit function call for the next one. Furthermore, if the underlying compiler optimizes recursive terminal calls, CPS completely suppress the "normal" implicit language stack, avoiding error such as the overflow of the stack.

Since the step-by-step execution is an objective of this dissertation, this full control over program flow will be useful. The continuation function can be used as a block, thus allowing to stop and return the execution.

Automatic CPS Transformation. As part of the FACTOR project objectives, an extension (ppx) incorporated in the OCaml compilation process has been developed. This extension uses technologies to modify code and syntactic expressions to automatically rewrite code into CPS [63].

3.4 Defunctionalization

Defunctionalization is a program transformation technique to convert high-order programs into first-order. Originally, it was introduced by Reynolds as a technique to transform a higher-order interpreter into a first-order one [62]. Recent studies uses this technique to derive abstract machines for different strategies of evaluation of the lambda-calculus from compositional interpreters [3, 57].

Let us consider the example of our factorial function in CPS to better understand the defunctionalization technique:

```
let rec fact_cps (x: int) (k: int → int) : int
  = if x = 0 then k 1
    else fact_cps (x - 1) (fun arg → k (x * arg))
```

As it is possible to observe, there are two anonymous functions in the fact_cps function. The continuation `arg` and the identity function `fun x → x`. In order to defunctionalize this function, we need to represent in first-order these two anonymous functions. For that, we create a new type that captures the values of the free variables:

```
type 'a k =
    | Kid
    | KFact ('a k) (int)
```

The Kid represents the identity function, thus does not have any free variable. The KFact represents the function (fun arg → ...) having the continuation `k` and the value of x (int). Having this representation, it is possible to substitute the anonymous functions with the corresponding constructor:

```
let rec fact_cps (x: int) (k: int → int) : int
  = if x = 0 then ...
    else fact_cps (x - 1) (KFact k x)
```

The next step of the process is to substitute the applications in the original program. For that, we introduce the `apply` function:

```
let rec apply (k: 'a k) (v: int)
  = match k with
    | Kid → v
    | KFact k x → apply k (x * v)
    end
```

The final step is to use this function to replace all the applications of the continuation `k`:

```
let rec fact_cps (x: int) (k: int → int) : int
  = if x = 0 then apply k 1
    else fact_cps (x - 1) (KFact k x)
```

Transformation process. As seen above, the defunctionalization transformation follows two general steps, allowing thus the mechanization of this process: get a first order representation of the function continuations, replace the continuations with this new representation and then introduce a new a function `apply` which replaces the applications of functions in the original program.

Chapter 4

Supporting Boolean Theories

A rich Boolean theory allows us to have clear proofs and increase the degree of automation. Since we want to have control and more in-depth knowledge of the theory, we preferred to adopt a back-to-basics strategy and build ourselves the Boolean theory that will support our implementations.

This chapter presents the foundation of our work. Section 4.1 exhibits our Boolean theory, and in Sections 4.2 and 4.3 the theories for Boolean and positive literals sets, respectively.

4.1 Boolean Theory

In a Boolean Theory or Boolean Algebra the values of the underlying set are true and false. It is a formal way for describing logical operations in the same way that elementary algebra describes numerical operations.

A Boolean Algebra consists of a set S, equipped with two binary operations (conjunction and disjunction), one unary operation (negation) and two elements (bot and top). To implement this set in Why3, we first define the type `t` with the `bot` and `top` constants:

```
type t
constant bot: t
constant top: t
```

For the operations, we implement them as functions, resorting the main ones to their respectives of the Why3 Boolean type. The `/*\` function defines conjunction (using the conjunction of Why3), the `\*/` function defines disjunction (using the disjunction of Why3), then `neg` function defines negation as a complement operation, and additionally, the `->*` function defines implication as an abbreviation, the composition of disjunction and complement operations, as usual. The code follows.

```
let function (/*\) (x y : t) : t  =
  if x = top ∧ y = top then top else bot

function (\*/) (x y : t) : t  =
  if x = top  ∨  y = top then top else bot

function neg (x : t) : t  =
  if x = bot then top else bot

function (→*) (x y : t) : t =
  (neg x) \*/ y
```

Six axioms defines the set S. Table 4.1 shows these properties and the corresponding Why3 Lemma.

Property Name	Operation	Boolean Property	Why3 Lemma
Absorption	Conjunction	$a \wedge (a \vee b) = a$	(1)
	Disjunction	$a \vee (a \wedge b) = a$	(2)
Identity	Conjunction	$a \wedge \top = a$	(3)
	Disjunction	$a \vee \bot = a$	(4)
Associativity	Conjunction	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	(5)
	Disjunction	$a \vee (b \vee c) = (a \vee b) \vee c$	(6)
Commutativity	Conjunction	$a \wedge b = b \wedge a$	(7)
	Disjunction	$a \vee b = b \vee a$	(8)
Distributivity	Conjunction	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	(9)
	Disjunction	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	(10)
Complements	Conjunction	$a \wedge \neg a = \bot$	(11)
	Disjunction	$a \vee \neg a = \top$	(12)

Table 4.1: Properties of Boolean Algebra and correspondent Why3 Lemma.

(1)
```
lemma and_abso_elem: forall x. x /*\ bot = bot
```

(2)
```
lemma or_abso_elem: forall x. x \*/ top = top
```

(3)
```
lemma and_neutral_elem: forall x. x /*\ top = x
```

(4)
```
lemma or_neutral_elem: forall x. x \*/ bot = x
```

(5)
```
lemma and_assoc: forall x y z. x /*\ (y /*\ z) = (x /*\ y) /*\ z
```

(6)
```
lemma or_assoc: forall x y z. x \*/ (y \*/ z) = (x \*/ y) \*/ z
```

(7)
```
lemma and_comm: forall x y : t. x /*\ y = y /*\ x
```

(8)
```
lemma or_comm: forall x y : t. x \*/ y = y \*/ x
```

(9)
```
lemma and_distr:
    forall x y z : t. x /*\ ( y \*/ z) = (x /*\ y ) \*/ (x /*\ z)
```

(10)
```
lemma or_distr:
    forall x y z : t. x \*/ (y /*\ z) = (x \*/ y) /*\ (x \*/ z)
```

(11)
```
lemma compl_bot: forall x : t. x /*\ neg x = bot
```

(12)
```
lemma compl_top: forall x : t. x \*/ neg x = top
```

If we define $a + b := (a \wedge \neg b) \vee (b \wedge \neg a) = (a \vee b) \wedge \neg(a \wedge b)$ and $a \,.\, b := a \wedge b$, the Boolean Algebra induces a Boolean ring (a ring that holds the property $a^2 = a$). The zero element of the ring coincides with the $\bot$ of the Boolean Algebra, and the multiplicative identity element with the $\top$ [65].

We also ensure the properties of the Table 4.2 hold the Boolean Algebra.

Property Name		**Boolean Property**	**Why3 Lemma**
Top is equivalent to the negation of bot		$\top = \neg\bot$	(1)
Double negation		$\neg\neg a = a$	(2)
De Morgan's Laws	Conjunction	$\neg(a \wedge b) = (\neg a \vee \neg b)$	(3)
	Disjunction	$\neg(a \vee b) = (\neg a \wedge \neg b)$	(4)

Table 4.2: Additional properties of Boolean Algebra and correspondent Why3 Lemma.

(1)
```
lemma repr_of_top : (top) = (neg (bot))
```

(2)
```
lemma doubleneg: forall b. neg (neg b) = b
```

(3)
```
lemma deMorgan_and: forall x1 x2. neg (x1 /*\ x2) = ((neg x1) \*/ (neg x2))
```

(4)
```
lemma deMorgan_or: forall x1 x2. neg (x1 \*/ x2) = ((neg x1) /*\ (neg x2))
```

We could also define the property that ensures that bot differs from top, but since that would not give more information to the provers, we decide to omit it.

4.2 Boolean Sets Theory

Algorithms commonly use sets as their primary data type. The transformation algorithm from Conjunctive Normal Form to Horn Clauses, presented in Chapter 6, uses them to keep track of the negative literals of certain formulae.

In this theory a range of supporting lemmas holds for a set of our Boolean type. We start by defining the set of type `t` using the fset module present in the standard library of Why3. This module provides functions to manipulate sets: `add` to insert elements in the set; `remove` that deletes elements from the set; `pick` that takes one element from the set; `cardinal` that returns the number of elements in the set; `is_empty` that checks for emptiness; and `empty`, the constant that represents an empty set.

```
type boolset = fset t
```

For the evaluation, we have distinguished two ways of evaluating each set. The first one evaluates each element of the set positively using the conjunction operator and the second one negatively using the disjunction operator:

```
let rec ghost function eval_positive (s: boolset): t
  variant { cardinal s }
= if is_empty s then neg bot
  else let x = pick s in
    x /*\ eval_positive (remove x s)

let rec ghost function eval_negative (s: boolset) : t
  variant { cardinal s }
= if is_empty s then bot
  else let x = pick s in
    neg (x) \*/ eval_negative (remove x s)
```

The fset module does not implement the `pick` function. It is a specification, therefore, it can only be used in a logical context (ghost).

Table 4.3 contains the definitions of the supporting lemmas, based on the evaluation functions. The first one tells that evaluating a set negatively is equivalent to the negation of the positive evaluation. The second and third lemmas, states that evaluating a set with an added element is equivalent to the value of the added element with the evaluation of the rest of the set. The last four are the absorbent and neutral elements. In the absorbent lemmas, adding `bot` to a set will give an evaluation of bot/top (depending on the evaluation function, bot if the positive and top if the negative) no matter what previous elements presented in. In the neutral lemmas, evaluating a set with an added `top` is equivalent to evaluating the set without it.

Property Name		**Boolean Property**	**Why3 Lemma**
Negative eval equivalent to negation of positive		$nval(s) = \neg pval(s)$	(1)
Adding element evaluation	Positive eval	$pval(s \cup \{a\}) = a \wedge pval(s)$	(2)
	Negative eval	$nval(s \cup \{a\}) = \neg a \vee nval(s)$	(3)
Adding absorvent element	Positive eval	$pval(s \cup \{\bot\}) = \bot$	(4)
	Negative eval	$nval(s \cup \{\bot\}) = \top$	(5)
Adding neutral element	Positive eval	$pval(s \cup \{\top\}) = pval(s \backslash \{\top\})$	(6)
	Negative eval	$nval(s \cup \{\top\}) = nval(s \backslash \{\top\})$	(7)

Table 4.3: Properties of Boolean set theory and correspondent Why3 Lemma.

(1)
```
let rec lemma neg_positive_isnegative (s: boolset)
  variant {cardinal s}
  ensures {neg (eval_positive s) = eval_negative s }
  = if cardinal s > 0 then
      let x = pick s in
        neg_positive_isnegative (remove x s)
```

(2)
```
lemma eval_positive_add: forall s x.
  eval_positive (add x s) = ((x) /*\ eval_positive s )
```

(3)
```
lemma eval_negative_add: forall s x.
  eval_negative (add x s) = ((neg (x)) \*/ eval_negative s )
```

(4)
```
lemma eval_positive_abso: forall s x.
  x = bot → (eval_positive (add x s) = bot)
```

(5)
```
lemma eval_negative_abso: forall s x.
  x = bot → (eval_negative (add x s) = top)
```

(6)
```
lemma eval_positive_neutral: forall s x.
  x = top → (eval_positive (add x s) = eval_positive s)
```

(7)
```
lemma eval_negative_neutral: forall s x.
  x = top → (eval_negative (add x s) = eval_negative s)
```

4.3 Theory of Sets of Positive Literals

As mentioned in the previous section, the algorithm presented in Chapter 6 uses a set to keep track of the negative literals of a specific formula. We now present a similar theory but, in this case, using a set of positive literals. A positive literal is a positive atomic

formula. Therefore, the type pliteral only contains a bottom element $\bot$ (denoting false) or variables.

```
type pliteral =
  | LBottom
  | LVar i

 type pliteralset = fset pliteral
```

To evaluate this type of set, we cast it to our Boolean set and later use its evaluation functions. The cast function is axiomatic and, given a set and a map from `i` to `t`, outputs a Boolean set.

```
function cast_setPF_setB (fset pliteral) (i → t) : fset t
```

The following two axioms define the function. The `cast_def_empty` tells that casting an empty set results also in empty set.

```
axiom cast_def_empty: forall s f. is_empty s → cast_setPF_setB s f = empty
```

In the case the set is not empty (the axiom cast_def_add), we construct the cast set recursively, picking one element at each iteration and adding the result of its evaluation to the set.

```
axiom cast_def_add : forall s f.
  not (is_empty s) → forall x. mem x s →
  cast_setPF_setB s f = add (eval_pliteral x f) (cast_setPF_setB (remove x s) f)
```

Note that, the `eval_pliteral` is a function that, using the map from `i` to `t`, converts the type pliteral to t.

The evaluation functions are then the following:

```
let rec ghost function eval_negative (s: fset pliteral) (f: i → t) : t
 = BoolSet.eval_negative (cast_setPF_setB s f)

 let rec ghost function eval_positive (s: fset pliteral) (f: i → t) : t
 = BoolSet.eval_positive (cast_setPF_setB s f)
```

For the lemmas, we ensure the same properties of the Boolean set theory (Table 4.3) and one added property. This apprises that evaluating a pliteral element is the same as evaluating a set with only that element (singleton):

```
lemma eval_singleton_equalEvalpliteral:
  forall x e1 e2 f. e1 = CPL x ∧ e2 = singleton x →
    eval_positive e2 f = Horn.eval_positive e1 f
```

Chapter 5

Transformation Algorithm to Conjunctive Normal Form

The Conjunctive Normal Form (CNF) [1] is commonly used in logical algorithms. The algorithm for converting propositional formulae to CNF is often presented formally, with rigorous mathematical definitions that are sometimes difficult to read [26, 32, 51], or informally, intended for Computer Science but with textual definitions in non-executable pseudo-code [13, 36]. The implementation of algorithms of this nature is a fundamental piece for learning and understanding them. Herein, we present its implementation, formally verified in Why3, from a presentation as a recursive function of the conversion algorithm to CNF.

5.1 Functional presentation of the algorithm

For simplicity let us call T to the algorithm that converts any propositional logic formula to CNF. A propositional logic formula ϕ is an element of the set G_p, defined as follows:

$$\begin{aligned} G_p \triangleq\ & \phi ::= \text{ł} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi && \textit{(formula)} \\ & \text{ł} ::= p \mid \bot && \textit{(atomic\_formula)}, \end{aligned}$$

where p ranges over a set of propositional variables.

The function T produces formulae in CNF, where a formula in CNF is an element of

[1] A formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals and a literal is a propositional symbol or its negation.

the set J_p, defined as follows:

$$\begin{aligned} J_p \triangleq\ & \chi ::= \chi \wedge \chi \mid \tau \\ & \tau ::= l \mid \neg l \mid \tau \vee \tau \\ & \text{ł} ::= p \mid \bot \end{aligned}$$

Herein, we have T: $G_p \rightarrow J_p$, where:

$$\mathrm{T}(\phi) = \text{cnfc (nnfc (impl\_free } (\phi)))$$

The algorithm composes three functions:

- The `impl_free` function, responsible for eliminating the implications;
- The `nnfc` function, responsible for converting to Negation Normal Form (NNF). A formula is in NNF if the negation operator is only applied to sub-formulae that are literals.
- The `cnfc` function, responsible for converting from NNF to CNF.

Each of the functions produces propositional formulae from different sets. The `cnfc` function produces formulae from the J_p set previously defined. The `impl_free` function produces formulae from the H_p set and the `nnfc` function from the I_p set:

$$\begin{aligned} H_p \triangleq\ & \psi ::= \text{ł} \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \\ & \text{ł} ::= p \mid \bot \end{aligned}$$

$$\begin{aligned} I_p \triangleq\ & \epsilon ::= \text{ł} \mid \neg\text{ł} \mid \epsilon \wedge \epsilon \mid \epsilon \vee \epsilon \\ & \text{ł} ::= p \mid \bot \end{aligned}$$

5.2 Implementation

The first step in the implementation is to define the types of the formulae according to the grammars presented in the previous section.

Analysing the sets, it is possible to observe that the grammar of atomic formulae (literals) is present in all of them. So, we decided to create a general type `literal` representing this grammar:

```
type pliteral =
    | LBottom
    | LVar i
```

The set G_p has literals, conjunctions, disjunctions, implications, and a primitive negation connective. It is represented by the type `formula`:

```
type formula =
  | L pliteral
  | Neg formula
  | And formula formula
  | Or formula formula
  | Impl formula formula
```

The set H_p is the set G_p without implications, and is represented by the type `formula_wi`:

```
type formula_wi =
  | L_wi pliteral
  | FAnd_wi formula_wi formula_wi
  | FOr_wi formula_wi formula_wi
  | FNeg_wi formula_wi
```

The set I_p has literals, conjunctions, disjunctions, and the negation connective (applied to literals). It is represented by the type `formula_nnf`:

```
type formula_nnf =
  | L_nnf pliteral
  | FNeg_nnf pliteral
  | FAnd_nnf formula_nnf formula_nnf
  | FOr_nnf formula_nnf formula_nnf
```

The set J_p has literals, negation of literals, disjunctions, and conjunctions. The conjunctions are only at the top, that means that after a disjunction there is not possible to find any conjunction. This set is represented by the type `formula_cnf`:

```
type formula_cnf =
  | FClause_cnf clause_cnf
  | FAnd_cnf formula_cnf formula_cnf

type clause_cnf =
  | DLiteral pliteral
  | DNeg_cnf pliteral
  | DOr_cnf clause_cnf clause_cnf
```

Functions. The function `impl_free` removes all implications. It is recursively defined as homomorphic in all cases, except in the implication case where it takes advantage of the Propositional Logic Law:

$$A \rightarrow B \equiv \neg A \vee B$$

It converts the constructions of the type `formula` for those of the type `formula_wi` and does recursive calls over the arguments:

```
let rec impl_free (phi: formula) : formula_wi
  = match phi with
    | L  phi → L_wi phi
    | Neg phi1 → FNeg_wi (impl_free phi1)
    | Or phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
    | And phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
    | Impl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
    end
```

The functions `nnfc` converts formulae to NNF. It is recursively defined over a combination of constructors: applying the Propositional Logic Law $\neg\neg A \equiv A$ the double negations are eliminated and using the De Morgan Laws, negations of conjunctions become disjunction of negations and negations of disjunctions become conjunction of negations. The code of the function is as follows:

```
let rec nnfc (phi: formula_wi) : formula_nnf
  = match phi with
    | L_wi phi1 → L_nnf phi1
    | FNeg_wi (FNeg_wi phi1) → nnfc phi1
    | FNeg_wi (FAnd_wi phi1 phi2) →
        FOr_nnf (nnfc (FNeg_wi phi1)) (nnfc (FNeg_wi phi2))
    | FNeg_wi (FOr_wi phi1 phi2) →
        FAnd_nnf (nnfc (FNeg_wi phi1)) (nnfc (FNeg_wi phi2))
    | FNeg_wi (L_wi phi1) → FNeg_nnf (phi1)
    | FOr_wi phi1 phi2 → FOr_nnf (nnfc phi1) (nnfc phi2)
    | FAnd_wi phi1 phi2 → FAnd_nnf (nnfc phi1) (nnfc phi2)
    end
```

The `cnfc` function converts formulae from NNF to CNF. It is straightforwardly defined except in the disjunction case, where it distributes the disjunction by the conjunction calling the auxiliary function `distr`.

```
let rec cnfc (phi: formula_nnf) : formula_cnf
  = match phi with
    | L_nnf literal → FClause_cnf (DLiteral literal)
    | FOr_nnf phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
    | FAnd_nnf phi1 phi2 → FAnd_cnf (cnfc phi1) (cnfc phi2)
    | FNeg_nnf literal → FClause_cnf (DNeg_cnf literal)
    end
```

The `distr` function uses the Propositional Logic Law

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C),$$

the code being the following:

```
let rec distr (phi1 phi2: formula_cnf) : formula_cnf
  = match phi1, phi2 with
    | FClause_cnf phi1, FClause_cnf phi2 → FClause_cnf (DOr_cnf phi1 phi2)
    | FAnd_cnf phi11 phi12, phi2 →
          FAnd_cnf (distr phi11 phi2) (distr phi12 phi2)
    | phi1, FAnd_cnf phi21 phi22 →
          FAnd_cnf (distr phi1 phi21) (distr phi1 phi22)
    end
```

Lastly, the code of the function (T) composes all of these functions:

```
let t (phi: formula) : formula_cnf
  = cnfc(nnfc(impl_free phi))
```

The whole implementation is in Appendix A.1.

5.3 How to obtain the correctness

Since the T algorithm is a composition of three functions, the correctness of the algorithm is the result of the correctness criteria of each of these three functions.

Criteria. The defined types represent exactly the grammar, so the equivalence of the input and output formula is the only criterion needed to ensure the verification. The evaluation functions for each type ensures this criterion.

Semantics of formulae. Since the basic criterion of correctness is the logical equivalence of formulae, we need a function to assign a semantic to them. For that, we created the eval function:

```
type valuation = i → t

function eval_pliteral (l: pliteral) (f: valuation) : t
  = match l with
      | LBottom → bot
      | LVar i → f i
    end

function eval (phi: formula) (f: valuation) : t
  = match phi with
    | L e       → eval_pliteral e f
    | FAnd e1 e2  → eval e1 f /*\ eval e2 f
    | FOr e1 e2   → eval e1 f \*/ eval e2 f
    | FImpl e1 e2 → (eval e1 f →* eval e2 f)
    | FNeg e      → neg (eval e f)
    end
```

This function takes an argument of type `valuation` assigning a value of type `t`[2] to each variable of the formula, receives the formula to evaluate and returns a value of type `t`. For the base constructor, if `L` is a literal, the Boolean value of the variable or the value of the constant, respectively, are returned. For the remaining constructor cases, the associated formulae are recursively evaluated and the result translated into the corresponding operation of our Boolean theory. The evaluation function for the type of `formula_wi` is similar.
The evaluation functions for the remaining types are in Appendix A.2.

5.4 Proof of correctness

The proof of correctness consists in demonstrating that each function respects the correctness criteria defined in the previous section. We show, herein, the WhyML code accepted by Why3 as correct.

Correctness of impl_free. The equivalence of the formulae is ensured using the formula evaluation functions and we use the input formula as a measure to ensure termination.

```
let rec impl_free (phi: formula) : formula_wi
    variant{ phi }
    ensures{ forall v. eval v phi = eval_wi v result }
  = ...
```

Correctness of nnfc. In the proof of correctness it is not possible to use the formula itself as a measure of termination, since in the case of the distribution of negation by conjunction or disjunction, constructors are added to the head, making the structural inductive criterion not applicable. Hence, we define a function that counts the number of constructors of each formula and use it as termination measure:

```
function size (phi: formula_wi) : int
  = match phi with
      | FVar_wi _ | FConst_wi _ → 1
      | FNeg_wi phi → 1 + size phi
      | FAnd_wi phi1 phi2 | FOr_wi phi1 phi2 → 1 + size phi1 + size phi2
  end
```

To ensure the number of constructors can never be negative, we use the `size_nonneg` auxiliary lemma:

[2] t is our Boolean type

```
let rec lemma size_nonneg (phi: formula_wi)
  variant { phi }
  ensures { size phi ≥ 0 }
= match phi with
  | FVar_wi _ | FConst_wi _ → ()
  | FNeg_wi phi → size_nonneg phi
  | FAnd_wi phi1 phi2 | FOr_wi phi1 phi2 →
      size_nonneg phi1; size_nonneg phi2
end
```

Furthermore, with the termination measure defined, we can close the proof of correctness of the `nnfc` function:

```
let rec nnfc (phi: formula_wi)
    variant{ size phi }
    ensures{ (forall v. eval_wi v phi = eval_nnf v result) }
  = ...
```

Correctness of cnfc. This correctness proof is similar to the previous one:

```
let rec cnfc (phi: formula_wi)
    ensures{ (forall v. eval_nnf v phi = eval_cnf v result) }
    variant  { phi }
  = ...
```

Since the `cnfc` function uses the auxiliary function `distr`, we also need to prove its correctness. In this correctness proof we use a combination of evaluation functions to ensure the partial proof and a sum of size functions applied to both arguments to ensure the total proof:

```
let rec distr (phi1 phi2: formula_wi)
    ensures{ (forall v. ((eval_cnf v phi1 || eval_cnf v phi2) =
      eval_cnf v result)) }
    variant  { size phi1 + size phi2 }
  = ...
```

Correctness of t. With the proofs of correctness of each of the three functions performed, we can now obtain the proof of correctness of the function T:

```
let t (phi: formula) : formula_cnf
    ensures{ (forall v. eval v phi = eval_cnf v result)}
  = ...
```

The whole code of the specification is in Appendix A.3. Using the Why3 `session` shell command, one obtains a table with proof times for every sub-goal. This exhaustive result is not very informative. However, adding all sub-goals times represents a theoretical

worst case scenario where tasks would be proved sequentially (the real behaviour is in fact a parallel execution). Nevertheless, we will use this criterion when presenting the proof time results, as it makes easier a comparison and allow us to present more compact tables. Table 5.1 shows the time of aggregated proof obligation of the CNF conversion algorithm. We present the results of CVC4, Alt-Ergo and Z3, however, the last one times out trying to prove some verification conditions. For the next tables of proof times, we will only show the results for the provers that validates most of the verification condition.

Proof obligations		Alt-Ergo 2.2.0	CVC4 1.6	Z3 4.8.4
lemma VC for impl_free	lemma variant decrease	0.21	0.16	0.14
	lemma postcondition	4.42	0.26	
lemma VC for nnfc	lemma variant decrease	0.09	0.32	0.18
	lemma postcondition	0.29	0.15	
lemma VC for distr	lemma variant decrease	0.06	0.37	0.15
	lemma postcondition	0.38	0.24	
lemma VC for cnfc	lemma variant decrease	0.08	0.25	0.12
	lemma postcondition	0.04	0.24	
lemma VC for t		0.01	0.07	0.02

Table 5.1: Aggregated proof time of CNF proof obligations.

5.5 Conclusions and Observations

Classical logical algorithms presented as functions to undergraduates can have a very close functional implementation that is easy to prove correct with a high degree of automation. The implementation was proved sound with small effort, basically following from the assertions one naturally associates with the code to prove it correct.

However, undergraduates do not learn this algorithm using grammars. They learn it with two sets of formulae, the G_p and H_p, which increases the complexity of the proof. We first started following the structure of the algorithm that is present in the slides available to students. This first work was more complex, but corresponds exactly to the version that students learn, which may be an added value. Therefore, we present below this version.

Previous implementation and proof of correctness. The implementation is similar to the one presented in the sections above, but only with the G_p and H_p sets, so the signature of the `nnfc`, `cnfc` and `distr` functions must be according to the sets:

```
let rec nnfc (phi: formula_wi) : formula_wi = ...
```

```
let rec cnfc (phi: formula_wi) : formula_wi = ...

let rec distr (phi1 phi2: formula_wi) : formula_wi = ...

let t: (phi: formula) : formula_wi = ...
```

Proof of correctness. Only with the type `formula_wi` it is not possible to ensure the Normal Negation Form and Conjunctive Normal Form. So to ensure the NNF and CNF, we create two well-formed predicates. The `wf_negation_of_literals` predicate ensures that the negation connective is applied only to literals:

```
predicate wf_negations_of_literals (f: formula_wi)
  = match f with
    | FNeg_wi f → (match f with
                   | FOr_wi _ _ | FAnd_wi _ _ | FNeg_wi _ → false
                   | _ → wf_negations_of_literals f
                   end)
    | FOr_wi f1 f2
    | FAnd_wi f1 f2 →
        wf_negations_of_literals f1 ∧ wf_negations_of_literals f2
    | FVar_wi _ → true
    | FConst_wi _ → true
  end
```

The `wf_conjunctions_of_disjunctions` predicate ensures that the conjunctions are only at the top, so after a disjunction it is not possible to find any conjunction:

```
predicate wf_conjunctions_of_disjunctions (f: formula_wi)
  = match f with
    | FAnd_wi f1 f2 →
        wf_conjunctions_of_disjunctions f1 ∧ wf_conjunctions_of_disjunctions f2
    | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
    | FConst_wi _ → true
    | FVar_wi _ → true
    | FNeg_wi f1 → wf_conjunctions_of_disjunctions f1
  end

predicate wf_disjunctions (f: formula_wi)
  = match f with
    | FAnd_wi _ _ → false
    | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
    | FConst_wi _ → true
    | FVar_wi _ → true
    | FNeg_wi f1 → wf_disjunctions f1
  end
```

Using the `wf_negations_of_literals` predicate as postcondition it is possible to ensure the NNF:

```
let rec nnfc (phi: formula_wi) : formula_wi
    variant { size phi }
    ensures { (forall v. eval_wi v phi = eval_wi v result) }
    ensures { wf_negations_of_literals result }
  = ...
```

Using the `wf_conjunction_of_disjunctions` predicate it is possible to ensure the CNF:

```
 let rec cnfc (phi: formula_wi) : formula_wi
    requires { wf_negations_of_literals phi }
    variant { phi }
    ensures { (forall v. eval_wi v phi = eval_wi v result) }
    ensures { wf_negations_of_literals result }
    ensures { wf_conjunctions_of_disjunctions result }
  = ...

let rec distr (phi1 phi2: formula_wi) : formula_wi
    requires { wf_negations_of_literals phi1 ∧ wf_negations_of_literals phi2 }
    requires { wf_conjunctions_of_disjunctions phi1 ∧
      wf_conjunctions_of_disjunctions phi2 }
    variant { size phi1 + size phi2 }
    ensures { (forall v. eval_wi v (FOr_wi phi1 phi2) = eval_wi v result) }
    ensures { wf_negations_of_literals result ∧
      wf_conjunctions_of_disjunctions result }
  = ...
```

In this functions we have preconditions since the algorithm is a composition of three functions, so the input formula must be in NNF for the `cnfc` function and in NNF and CNF for the `distr`.

In the `distr` function, it is not possible to prove that a disjunction of two formulae in CNF is effectively a formula in CNF. We must ensure that in a disjunction of two formulae in CNF, the formulae do not contain the conjunction constructor. To accomplish this, we use an auxiliary lemma:

```
lemma aux: forall x. wf_conjunctions_of_disjunctions x ∧
  wf_negations_of_literals x ∧ not (exists f1 f2. x = FAnd_wi f1 f2) →
  wf_disjunctions x
```

Chapter 6

Transformation Algorithm from CNF to Horn Clauses

The Horn algorithm [34, 60] is a simple and easy solution to decide in polynomial complexity if a given propositional formula is satisfactory or contradictory. However, the algorithm only works for a particular set of formulae - the Horn Clauses. Common logic literature, unfortunately, does little emphasis in the implementation of the transformation algorithm from CNF to these Horn Clauses, presenting only its definitions. We could not find any functional presentation of this algorithm. Therefore, this chapter contributes with a functional presentation, with its implementations and soundness proof.

6.1 Algorithm Definition

A basic Horn clause is a disjunction of literals, where at most one occurs positively. So, there are only three possibilities for a basic Horn clause:

1. Does not have any positive literal (atomic formula).
2. Does not have any negative literal, being only one positive literal.
3. Does have negative literals and only one positive.

Therefore, it is possible to present any basic Horn Clause as an implication:

1. $L \equiv \top \rightarrow L$
2. $\bigvee_{i=1}^{n} \neg L_i \equiv (\bigwedge_{i=1}^{n} L_i) \rightarrow \bot$
3. $\bigvee_{i=1}^{n} \neg L_i \vee L \equiv (\bigwedge_{i=1}^{n} L_i) \rightarrow L$

Where L and L_i (for all i) are positive literals.

A propositional formula is a Horn clause if it is a basic conjunction of Horn clauses,

$$\phi = \bigwedge_{i=1}^{n} (C_i \rightarrow L_i)$$

where L_i are positive literals and $C_i = \top$ or $C_i = \bigwedge_{j=1}^{k_i} L_{i,j}$ The following grammar defines a Horn formula:

$$\begin{aligned}
\psi &::= \mu \mid \psi \wedge \psi && \textit{(horn_formula)} \\
\mu &::= \chi \rightarrow \omega && \textit{(basic_horn_formula)} \\
\chi &::= \top \mid \alpha && \textit{(leftside)} \\
\alpha &::= p \mid \bot \mid \alpha \wedge \alpha && \textit{(positive)} \\
\omega &::= p \mid \bot \mid \top && \textit{(rightside)}
\end{aligned}$$

6.2 Functional Presentation of the Algorithm

The algorithm converts to a conjunction of basic Horn clauses, given a specific formula ϕ in Conjunctive Normal Form that is defined in the Page 29. The main function is called `hornify` and has the following signature:

$$hornify : formula\_cnf \rightarrow horn\_formula$$

Precisely, the function traverses each sub-formula of the conjunctions and calls the `getBasicHorn` function:

$$hornify(\phi) \triangleq \begin{cases} hornify(\phi_1) \wedge hornify(\phi_2), & if\ \phi = \phi_1 \wedge \phi_2 \\ getBasicHorn(\phi), & if\ otherwise \end{cases}$$

The function `getBasicHorn` converts propositional formulae in CNF without conjunctions (τ) into basic Horn clauses:

$$getBasicHorn : formula\_cnf \rightarrow basic\_horn\_formula$$

$$getBasicHorn(\phi) \triangleq \begin{cases} let\ (s,p) = hornify\_aux\ (\phi,\ \varnothing,\ \varnothing)\ in & \\ buildConjunction\ (s) \rightarrow getPositive\ (p), & if\ \phi = \phi_1 \vee \phi_2 \\ \phi_1 \rightarrow \bot, & if\ \phi = \neg\phi_1\ and\ \phi_1\ is\ a\ variable \\ \top \rightarrow \phi_1, & if\ \phi = \phi_1\ and\ \phi_1\ is\ a\ variable \\ \top \rightarrow \top, & if\ \phi = \neg\bot \\ \top \rightarrow \bot, & if\ \phi = \bot \end{cases}$$

This functions follows the properties presented in Section 6.1 to transform each disjunction or literal into an implication (basic Horn clause). The main case of the function is

when the formula is a disjunction. In this case, the function calls the `hornify_aux` function to get all the positive and negative literals into a set, then uses the `buildConjunction` function to build the conjunction of the negative literals and the `getPositive` function to get the positive literal. In the other cases the transformation is straightforward.

The function `hornify_aux` goes trought the formula and adds negative literals to the left set and positive literals to the right set:

$$hornify\_aux : disjunction * set * set \rightarrow set * set$$

$$hornify\_aux(\phi, s, p) \triangleq \begin{cases} let\ (s1, p1) = hornify\_aux\ (\phi_1,\ s,\ p)\ in & \\ hornify\_aux\ (\phi_2,\ s1,\ p1), & \text{if } \phi = \phi_1 \vee \phi_2 \\ (s \cup \{\phi_1\},\ p), & \text{if } \phi = \neg\phi_1 \\ (s, \{\phi\}), & \text{if } \phi \text{ is a variable and } p = \varnothing \end{cases}$$

The `buildConjunction` function returns a positive literal if the set has only one literal or constructs a conjunction with all the positive literals in the set:

$$buildConjunction : set \rightarrow leftside$$

$$buildConjunction(s) \triangleq \begin{cases} \phi, & \text{if } \phi \in s \text{ and } |s| = 1 \\ \phi \wedge (buildConjunction\ (s \setminus \{\phi\})), & \text{if } \phi \in s \text{ and } |s| > 1 \end{cases}$$

The `getPositive` functions is responsible for building the right-hand side of the implication. Given an empty set or a set with only one positive literal, it returns one positive literal (or $\varnothing$ or itself):

$$getPositive : set \rightarrow rightside$$

$$getPositive(p) \triangleq \begin{cases} \bot, & \text{if } p = \varnothing \\ \phi, & \text{if } p = \{\phi\} \end{cases}$$

6.3 Implementation

It is necessary to define the specific formulae types according to the grammar defined in Section 6.1. The Table 6.1 shows the correspondent Why3 type for each grammar.

Grammar Name	Grammar	Why3 Type
horn_clause	$\psi ::= \mu \mid \psi \wedge \psi$	(1)
basic_horn_clause	$\mu ::= \chi \rightarrow \omega$	(2)
leftside	$\chi ::= \top \mid \alpha$	(3)
positive	$\alpha ::= p \mid \bot \mid \alpha \wedge \alpha$	(4)
rightside	$\omega ::= p \mid \bot \mid \top$	(5)

Table 6.1: Why3 types according to grammars.

(1)
```
type horn_clause =
  | HBasic basic_horn_clause
  | HAnd horn_clause horn_clause
```

(2)
```
type basic_horn_clause =
  | BImpl leftside rightside
```

(3)
```
type leftside =
  | LTop
  | LPos positive
```

(4)
```
type positive =
  | PLCBottom
  | PLCVar ident
  | PLCAnd positive positive
```

(5)
```
type rightside =
  | RBottom
  | RTop
  | RVar ident
```

Functions. The implementation of the described functions follows the structure of the mathematical definitions. The main function (`hornify`) traverses the formula and calls the `getBasicHorn` function when the formula is a disjunction (`FClause_cnf`):

```
let rec hornify (phi: formula_cnf) : horn_clause
  = match phi with
    | FClause_cnf phi1 → HBasic (getBasicHorn phi1)
    | FAnd_cnf phi1 phi2 → HAnd (hornify phi1) (hornify phi2)
  end
```

The `getBasicHorn` converts disjunctions to an equivalent implication. The base cases use the conversion rules previously shown. In the inductive case (when the input formula

has disjunction as head operator) we use the auxiliary function `hornify_aux` (in the next page) in order to get all the positive and negative literals. This is a partial function, as by definition of Horn clause, there is at most one positive literal in a clause. Why3, in contrast of similar proof tools, supports functions with side effects. This comes in handy, allowing us to traverse each formula a single time: if we find two positive literals, we raise an exception (using a functional `option` type). The alternative would be to traverse it twice, the first to count the number of positive literals and the second to convert. Note that the `hornify_aux` function has three arguments: the formula, the set collecting the negative literals, and a value of the type `option` (the value `None`). Then we call the `buildConjunction` and `getPositive` functions to build the implication.

```
let getBasicHorn (phi: clause_cnf) : basic_horn_clause
  = match phi with
    | DLiteral (LVar x) → BImpl (LTop) (RVar x)
    | DLiteral (LBottom) → BImpl (LTop) (RProp bot)
    | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RProp bot)
    | DNeg_cnf (LBottom) → BImpl (LTop) (RProp top)
    | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
                          BImpl (buildConjunction s) (getPositive p)
    end
```

The implementation of `buildConjunction`, in turn, uses an auxiliary function `build` that returns a positive literal if the set has only one literal or constructs a conjunction with all the positive literals in the set. Since we want to extract the code, we use the applicative sets module (AppSet) from Why3 standard library. This module instead of the specification `pick` function, provides an executable function named `choose`. The function `convertPLtoPLC` transform constructors of the type `pliteral` to the corresponding ones of the type positive.

```
let buildConjunction (s: set): leftside
  = let rec build (s: set)
     = if(is_empty s) then absurd else
       if((cardinal s) = 1) then (convertPLtoPLC (choose s)) else
         let element = choose s in
          PLCAnd (convertPLtoPLC (element)) (build (remove (element) s)) in
    LPos (build s)
```

The `getPositive` function returns a positive literal: $\bot$ if the `option` is `None` or `x` if it is `Some x`.

```
 let getPositive (p: option rightside) : rightside
  = match p with
    | None → RProp bot
    | Some x → x
    end
```

The `hornify_aux` is a partial function that goes through the disjunction combinations, adds the negative literals to the input set (the literals are added as positive literals dropping the negative connector) and the positive literal to the `option` type.

This function, as previously said, can also raise an exception, called `MoreThanOnePositive`, if there is more than one positive literal in the formula. This exception is an identifier defined in the `hornify` module (Appendix B.1). When the disjunction constructor has two positive literals, we raise an exception. If it has one positive and one negative, we call the processCombination function, that will check if the `option` is defined and add the negative to set. When the constructor has at least another disjunction in his arguments, we recursively call the `hornify_aux` function.

```
let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside)
  : (rs: set, rp: option rightside)
  raises{ MoreThanOnePositive }
  = match phi with
    | DOr_cnf (DLiteral _) (DLiteral _) → raise MoreThanOnePositive
    | DOr_cnf (DLiteral pl) (DNeg_cnf nl)
    | DOr_cnf (DNeg_cnf nl) (DLiteral pl) → processCombination pl nl s p
    | DOr_cnf (DNeg_cnf nl1) (DNeg_cnf nl2) →
        ((add (convertPLiteralToPL nl1) (add (convertPLiteralToPL nl2) s)), p)
    | DOr_cnf (DOr_cnf phi1 phi2) (DLiteral pl)
    | DOr_cnf (DLiteral pl) (DOr_cnf phi1 phi2) →
        match p with
          | None →
              hornify_aux (DOr_cnf phi1 phi2) s (Some (convertLiteralToR pl))
          | Some _ → raise MoreThanOnePositive
        end
    | DOr_cnf (DOr_cnf phi1 phi2) (DNeg_cnf nl)
    | DOr_cnf (DNeg_cnf nl) (DOr_cnf phi1 phi2) →
         hornify_aux (DOr_cnf phi1 phi2) (add (convertPLiteralToPL nl) s) p
    | DOr_cnf phi1 phi2 →
        let (s1,p1) = hornify_aux phi1 s p in hornify_aux phi2 s1 p1
    | _ → absurd
  end
```

The `processCombination` function analyses the case when the disjunction constructor has one positive and one "negative" literal. If the input `option p` is `None` then the `addLiterals` function is called, otherwise the `MoreThanOnePositive` exception is raised.

```
let processCombination (pl: pliteral) (nl: pliteral) (s: set)
  (p: option rightside) : (rs: set, rp: option rightside)
  raises{ MoreThanOnePositive }
  = match p with
    | None → addLiterals pl nl s p
    | Some _ → raise MoreThanOnePositive
    end
```

The `addLiterals` function inserts the "negative" literal into the set and defines the option `rp` as `Some x`, where `x` is the positive literal.

```
let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside)
  : (rs: set, rp: option rightside)
  = match pl with
    | LBottom → let rbottom = Some (RProp bot) in
                match nl with
                | LBottom → ((add (Prop_pl bot) s), rbottom)
                | LVar x → ((add (Var_pl x) s), rbottom)
                end
    | LVar x → let rvar = Some (RVar x) in
                match nl with
                | LBottom → ((add (Prop_pl bot) s), rvar)
                | LVar x → ((add (Var_pl x) s), rvar)
                end
 end
```

The whole code is given in Appendix B.1

6.4 Proof of correctness

Since the defined types represent exactly the grammar of the output of the algorithm, the equivalence of the input and output formula is the only criterion needed to ensure that the implementation is sound. The valuation functions for each type ensures this criterion (Apendix B.2).

Less trivial cases. In some cases, especially when the function has more than one argument and pairs in the output, the equivalence can be a little tricky. A combination of valuation functions is needed to resolve these cases. In the `hornify_aux` function it is need to ensure that:

1. the output set has at least one literal (is not empty)
2. phi $\vee$ $(\bigvee_{i=1}^{n} \neg E_i) \vee$ p $\equiv$ $(\bigvee_{i=1}^{n} \neg R_i) \vee$ rp
3. phi $\vee$ $(\bigvee_{i=1}^{n} \neg E_i) \vee$ p $\equiv$ $(\bigwedge_{i=1}^{n} R_i) \rightarrow$ rp,

where phi is a disjunction, E_i are the elements of the input set, p is the input `option`, R_i are the elements of the output set and `rp` the output `option`.

This function only receives disjunctions; the other cases are absurd. Given the need to prove the absurd cases, and since the `clause_cnf` type contains non-disjunction constructors, we must ensure that the input formula (`phi`) is indeed a disjunction. This obligation is ensured by adding a precondition in the specification of the function.

```
let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) :
  (rs: set, rp: option rightside)
    requires{ exists x y. phi = DOr_cnf x y }
    ensures{ not is_empty rs }
    ensures{ forall f. eval_domain phi s p f = eval_codomain rs rp f }
    ensures{ forall f. eval_domain phi s p f = ((eval_positive rs f) →*
                (eval_optionrightside rp f)) }
    variant{ phi }
    ...
  = match phi with
    ...
    | _ → absurd
    end
```

The `eval_domain` evaluates the domain of the function (left side of the equality of property 2 and 3) and the `eval_codomain` the codomain (right-hand side of the equality of property 2):

```
function eval_domain (phi: clause_cnf) (s: set) (p: option rightside)
  (f: i → t) : t
  = eval_disj_cnf phi f \*/ eval_negative s f \*/ eval_optionrightside p f

function eval_codomain (s: set) (p: option rightside) (f: i → t) : t
  = eval_negative s f \*/  eval_optionrightside p f
```

In the `processCombination` and `addLiterals` functions one needs to ensure, once again, that the output set is not empty and that the domain is equivalent to the codomain. This obligation can be translated into the following property

$$pl \vee \neg nl \vee (\bigvee_{i=1}^{n} \neg E_i) \vee p \equiv (\bigvee_{i=1}^{n} \neg R_i) \vee rp,$$

where pl and nl are the positive literal and negative literal, respectively, E_i are the elements of the input set, p is the input `option`, R_i are the elements of the output set and rp the output `option`. The specification is the following one:

```
let processCombination (pl: pliteral) (nl: pliteral) (s: set)
  p: option rightside) : (rs: set, rp: option rightside)
    raises{ MoreThanOnePositive }
    ensures{ (not is_empty rs) }
    ensures { forall f.
      (eval_literal pl f \*/ (neg (eval_literal nl f)) \*/
      eval_negative s f \*/ eval_optionrightside p f) =
      (eval_negative rs f \*/ eval_optionrightside rp f) }
  = ...
```

The full specification is in Appendix B.3 and Table 6.2 shows the proof time of each proof obligation of our Horn Clause conversion algorithm. Once again, the times are an aggregation of every sub-goal proof time.

Proof obligations		CVC4 1.6
lemma VC for convertLiteralToR		0.09
lemma VC for addLiterals		1.08
lemma VC for processCombination		0.05
lemma VC for hornify_aux	lemma precondition	0.36
	lemma postcondition	2.53
	lemma variant decrease	0.60
	lemma unreachable point	0.09
lemma VC for buildConjunction	lemma precondition	0.23
	lemma variant decrease	0.11
	lemma postcondition	0.35
lemma VC for getPositive		0.09
lemma VC for getBasicHorn	lemma postcondition	1.47
	lemma precondition	0.06
lemma VC for hornify		0.13

Table 6.2: Aggregated proof time of Hornify proof obligations

6.5 Conclusions and Observations

We present herein a new formulation of the transformation from CNF formulae into Horn clauses. It is presented as recursive functions, being clear, readable, rigorous and ideal to undergraduate logic courses. However, even if the presentation is clear and fits on one page, the implementation and verification are not trivial.

The verification process did not escape the rule of only ensuring the equivalence of the domain and codomain evaluations. However, the domain and codomain presented in this chapter usually have more than one argument, turning the equivalence not trivial. We explicitly define the properties of equivalence and traduced it to a post-condition in Why3.

The theories of sets presented in Section 4.3 and 4.2 had a major rule in this proof. The type `set` of Why3 does not provides enough information, these hints gave them enough to again naturally process the proof.

Chapter 7

Towards Step-by-Step Execution

The ability to execute in step-by-step or even to rewind or step back computations is fundamental for programmers, as it permits the inspection of the intermediate values of computations. Moreover, forward and backwards stepwise execution provides a better understanding of the code under inspection. Previous work, also within the scope of the FACTOR project ("Rewinding functions through CPS"), shows how to support tracing functionalities in continuation-passing style programming [28]. To make use of the functionalities provide by the mentioned work, we developed versions of our implementations with explicit stack structure. Henceforth, this chapter presents the CPS/Defunctionalization transformation of the implementation listed in Chapter 5.

This transformation and its verification can be made automatically [63]. However, this work was released after the manual transformation listed in this chapter. The algorithm in Chapter 6 and future implementations will later be automatically transformed into CPS.

7.1 Continuation-Passing Style

As mentioned in Section 3.3, *Continuation-Passing Style* (CPS) is a programming style where the control is passed explicitly in the form of a continuation. We obtain the CPS version of the functions in a similar way to the process presented in Section 3.3. So, using the process we have the following code for the CPS version of the `impl_free` function:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
= match phi with
  | Prop t → if t = bot then k (L_wi (LBottom))
             else k (FNeg_wi (L_wi LBottom))
  | Var i → k (L_wi (LVar i))
  | Neg phi1 → impl_free_cps phi1 (fun processed_phi1 →
      k (FNeg_wi processed_phi1))
  | Or phi1 phi2 → impl_free_cps phi1 (fun impl_left →
      impl_free_cps phi2 (fun impl_right →
        k (FOr_wi impl_left impl_right)))
  | And phi1 phi2 → impl_free_cps phi1 (fun impl_left →
      impl_free_cps phi2 (fun impl_right →
          k (FAnd_wi impl_left impl_right)))
  | Impl phi1 phi2 → impl_free_cps phi1 (fun impl_left →
      impl_free_cps phi2 (fun impl_right →
        k (FOr_wi (FNeg_wi impl_left) impl_right)))
  end

let impl_free_main (phi: formula) : formula_wi
  = impl_free_cps phi (fun x → x)
```

The code for the remaining function is in Appendix A.4).

Correctness criteria. One interesting aspect of the proof of correctness of the functions in CPS is the use of the corresponding function in direct-style as specification, since these ones are pure and total. Briefly, we simply assure that the result of the CPS functions is equivalent to the result of the functions in direct style.

For the `impl_free` function in CPS, it is enough to ensure that the result is equivalent to the result of the direct-style `impl_free` function applied to the continuation:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  variant { phi }
  ensures { result = k (impl_free phi) }
= ...
```

The specification of the function in direct style is then also applied to the function `main`, responsible for calling the CPS functions with the identity function as continuation:

```
let impl_free_main (phi: formula) : formula_wi
  ensures { forall v. eval v phi = eval_wi v result }
= ...
```

The specifications of the `nnfc` and `cnfc` functions in CPS are similar and are presented in Appendix A.5. The proof time for each generated proof obligation can be observed in the Table 7.1. Once again, if they were executed sequentially.

Proof obligations		Alt-Ergo 2.2.0	CVC4 1.6	Z3 4.8.4
lemma VC for impl_free_cps	lemma variant decrease	0.21	0.50	
	lemma postcondition	0.16	0.41	0.15
lemma VC for impl_free_main		0.01	0.05	0.02
lemma VC for nnfc_cps			0.05	0.46
lemma VC for nnfc_main		0.01	0.04	0.03
lemma VC for distr_cps		0.21	0.15	0.22
lemma VC for distr_main		0.01	0.06	0.03
lemma VC for cnfc_cps		0.24	0.77	0.22
lemma VC for cnfc_main		0.01	0.05	0.02
lemma VC for t_main		0.01	0.05	0.02

Table 7.1: Aggregated proof time of CNF-CPS proof obligations.

Observations. This implementation, uses types that represent grammars but, as mentioned in Section 5.5, undergraduates learn it only with two sets of formulae. Using these types, most correctness criterion are "automatically" ensured, because the functions output is tailored according to its properties. We only need to ensure the equivalence of the evaluation of the domain and codomain. However, with two sets of formulae we need to introduce well-formed predicates to ensure certain criteria. For example, the `wf_negations_of_literals` well-formed predicate ensures that a formula is in NNF.

These well-formed predicates increased the complexity of the CPS proof, as proof obligations are generated concerning the validity of pre-conditions whenever a recursive call is made within a continuation. In order to prove such a proof obligation, we need to specify the nature of the continuation arguments. Thus, we encapsulate the well-formed predicates into new types (invariant types). The following code represents the invariant type with the `wf_negations_of_literals` well-formed predicated encapsulated:

```
type nnfc_type = { nnfc_formula : formula_wi }
invariant { wf_negations_of_literals nnfc_formula }
by{ nnfc_formula = FConst_wi true }
```

With this, the return type of the functions has been changed to an invariant type rather than a normal type. So, the post-conditions now involves the comparison of two invariant types, which raises some interesting challenges.

Difficulties to achieve a proof. Comparing two invariant types involves providing them a witness, i.e., values with the concerned type; only then it is possible to prove that two values of the same type respect the invariant. However, as the invariant type in Why3 is an opaque type, having only access to its projections, it is not possible to construct an

inhabitant of this type in the logic, thus making it impossible to compare them. This lemma translates such a behaviour:

```
lemma types: forall x y. x.nnfc_formula = y.nnfc_formula → x = y
```

It is not possible to prove this lemma because having only access to record projections can not ensure that, in this case, the field `nnfc_formula` is the only field of this record type. Given this limitation of Why3 [2], which in this case precludes the proof of the post-condition, we have tried to compare the formula of each type with an extensional equality predicate (==) and use this predicate as post-condition instead of polymorphic structural equality (=).

```
predicate (==) (t1 t2: nnfc_type) = t1.nnfc_formula = t2.nnfc_formula
```

Even with extensional equality, it was not possible to complete the proof. This is due to the fact that for the base cases, given the application to the continuation, we always come across with comparison of records and in the other cases it is not possible to specify the functions of continuation in the recursive calls. This lack of success led to the search for other approaches that would, eventually, achieve the same advantages as the CPS transformation.

What is the problem with CPS?. The transformation in CPS always adds a function as an argument, thus passing to a higher-order function. Since Why3 is a platform that, for reasons of automation, operates on a first-order language, the solution is to "go back" to first-order. The defunctionalization technique emerged as a possible approach.

7.2 Defunctionalization

Defunctionalization is a program transformation technique to convert high-order programs into first-order ones [62]. It produces an evaluator, a version close to a first-order abstract machine [3].

Transformation process. Following the process presented in Section 3.4, a defunctionalization consists of a "mechanical" transformation in two steps:

1. Get a first order representation of the function continuations and replace the continuations with this new representation.

2. Generate a new function (`apply`) which replaces the applications of functions in the original program.

Applying this process to the `impl_free` function in CPS lead us to the following representation for the function continuations:

```
type impl_kont =
  | KImpl_Id
  | KImpl_Neg impl_kont
  | KImpl_OrLeft impl_kont formula
  | KImpl_OrRight impl_kont formula_wi
  | KImpl_AndLeft impl_kont formula
  | KImpl_AndRight impl_kont formula_wi
  | KImpl_ImplLeft impl_kont formula
  | KImpl_ImplRight impl_kont formula_wi
```

The constructor `KImpl_id` represents the identity function, and the constructor `KImpl_Neg` represents the continuation of the constructor `FNeg_wi`. The remaining cases contain two continuation functions, so two constructors are created, one `left` and one `right`. We chose to use the `left` and `right` nomenclatures because this represents the natural order of the formula in the abstract syntax tree.

We now replace the continuations with this new representation of the function continuations:

```
let rec impl_free_defun (phi: formula) (k: impl_kont) : formula_wi
= match phi with
  ...
  | Neg phi1 → impl_free_defun phi1 (KImpl_Neg k)
  | Or phi1 phi2 → impl_free_defun phi1 (KImpl_OrLeft k phi2)
  | And phi1 phi2 → impl_free_defun phi1 (KImpl_AndLeft k phi2)
  | Impl phi1 phi2 → impl_free_defun phi1 (KImpl_ImplLeft k phi2)
  end
```

Then we introduce the `apply` function, which replaces the applications of the continuation. This function is mutually recursive with the `impl_free_defun`. The identity constructor simply returns the formula argument, where the `KImpl_Neg` constructor recursively calls the `impl_apply` function with its argument `k` and the formula already processed (without implications) applied to the `FNeg_wi` constructor. For the remaining cases:

- If it is a `Left` "continuation" constructor, we call the `impl_free_defun` function with `phi2` (the right side formula of the corresponding head constructor); and the `Right` "continuation" constructor, applied to `k` – the continuation argument of the left constructor – and to `impl_left` – the left side already processed (without implications).

- If it is a `Right` "continuation" constructor, we recursively call the `impl_apply` function with two arguments: the application `k` and the result of applying the corresponding formula constructor to the `impl_left` and `impl_right`.

```
with impl_apply (k: impl_kont) (arg: formula_wi) : formula_wi = match k with
  | KImpl_Id → let x = arg in x
  | KImpl_Neg k → let processed_phi1 = arg in
      impl_apply k (FNeg_wi processed_phi1)
  | KImpl_OrLeft k phi2 → let impl_left = arg in
      impl_free_defun phi2 (KImpl_OrRight k impl_left)
  | KImpl_OrRight k impl_left → let impl_right = arg in
      impl_apply k (FOr_wi impl_left impl_right)
  | KImpl_AndLeft k phi2 → let impl_left = arg in
      impl_free_defun phi2 (KImpl_AndRight k impl_left)
  | KImpl_AndRight k impl_left → let impl_right = arg in
      impl_apply k (FAnd_wi impl_left impl_right)
  | KImpl_ImplLeft k phi2 → let impl_left = arg in
      impl_free_defun phi2 (KImpl_ImplRight k impl_left)
  | KImpl_ImplRight k impl_left → let impl_right = arg in
      impl_apply k (FOr_wi (FNeg_wi impl_left) impl_right)
  end
```

Finally, we replace the applications of the continuation with the `impl_apply` function:

```
let rec impl_free_defun (phi: formula) (k: impl_kont) : formula_wi
= match phi with
  | Prop t → if t = bot then impl_apply k (L_wi (LBottom))
             else impl_apply k (FNeg_wi (L_wi LBottom))
  | Var i → impl_apply k (L_wi (LVar i))
  ...
  end
```

The `impl_free_defun` is the result of the defunctionalization transformation of the `impl_free_cps` function.

The result of the application of the defunctionalization transformation to the remaining functions of the T algorithm in CPS is given in Appendix A.6.

Proof of correctness. The defunctionalized program specification is the same as the original program. However, given the existence of an additional function generated by the defunctionalization process (the `apply` function), a specification must be provided. Since the `apply` function simulates the application of a function to its argument, the only specification we can give it is that its post-condition is the post-condition of the function `k` [57].

To be able to use the direct-style functions as a specification, we have created a `post` predicate that gathers the post-conditions of the direct-style function. As for the `apply` function, such predicate performs case analysis on the continuation type; and for each constructor, we copy the post-condition present in the corresponding abstraction [57]. For example, the `impl_free_cps` function, has the following specification:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  ensures { result = k (impl_free phi) }
= ...
```

However, if we use the `post` predicate, we can change the post-condition to:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  ensures { post k (impl_free phi) result }
= ...
```

This post-condition establish a relationship between the value passed to the continuation k (a formula without implications) and the output (`result`). Following this methodology, we can, also, specify the anonymous functions used inside `impl_free_cps` function. The post-condition of the constructor `Neg`, compares `processed_phi1` applied to `FNeg_wi` with the `result` formula; this comparison depends on the continuation `k`. At the top level, the initial continuation is the identify function, hence the result will be the expected one. The remaining cases follows the same pattern.

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
    ensures{ post k (impl_free phi) result}
= match phi with
  | Prop t → if t = bot then k (L_wi (LBottom)) else k (FNeg_wi (L_wi LBottom))
  | Var i → k (L_wi (LVar i))
  | Neg phi1 →
      impl_free_cps phi1 (fun processed_phi1 →
        ensures { post k (KNeg_wi processed_phi1) result }
          k (FNeg_wi processed_phi1))
  | Or phi1 phi2 →
      impl_free_cps phi1 (fun impl_left →
        ensures { post k (FOr_wi impl_left (impl_free phi2)) result }
          impl_free_cps phi2 (fun impl_right →
            ensures { post k (FOr_wi impl_left impl_right) result }
              k (FOr_wi impl_left impl_right)))
  | And phi1 phi2 →
      impl_free_cps phi1 (fun impl_left →
        ensures { post k (FAnd_wi impl_left (impl_free phi2)) result }
          impl_free_cps phi2 (fun impl_right →
            ensures { post k (FAnd_wi impl_left impl_right) result }
              k (FAnd_wi impl_left impl_right)))
  | Impl phi1 phi2 →
      impl_free_cps phi1 (fun impl_left →
        ensures { post k (FOr_wi (FNeg_wi impl_left) (impl_free phi2)) result }
          impl_free_cps phi2 (fun impl_right →
            ensures { post k (FOr_wi (FNeg_wi impl_left) impl_right) result }
              k (FOr_wi (FNeg_wi impl_left) impl_right)))
  end
```

The post predicate, used in the proof of our defunctionalization version, gathers all of these anonymous function specifications according to its corresponding constructor. The applications of k are similar to the ones of the apply function, but, instead of defining the right-side "continuation" constructor, we call the direct-style impl_free function.

```
predicate impl_post (k: impl_kont) (arg result: formula_wi)
= match k with
  | KImpl_Id → let x = arg in x = result
  | KImpl_Neg k → let processed_phi1 = arg in
      impl_post k (FNeg_wi processed_phi1) result
  | KImpl_OrLeft k phi2 → let impl_left = arg in
      impl_post k (FOr_wi impl_left (impl_free phi2)) result
  | KImpl_OrRight k impl_left → let impl_right = arg in
      impl_post k (FOr_wi impl_left impl_right) result
  | KImpl_AndLeft k phi2 → let impl_left = arg in
      impl_post k (FAnd_wi impl_left (impl_free phi2)) result
  | KImpl_AndRight k impl_left → let impl_right = arg in
      impl_post k (FAnd_wi impl_left impl_right) result
  | KImpl_ImplLeft k phi2 → let impl_left = arg in
      impl_post k (FOr_wi (FNeg_wi impl_left) (impl_free phi2)) result
  | KImpl_ImplRight k impl_left → let impl_right = arg in
      impl_post k (FOr_wi (FNeg_wi impl_left) impl_right) result
  end
```

Finally, we use the impl_post predicate to specify the impl_free defunctionalized function:

```
let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
    ensures{impl_post k (impl_free phi) result}
  = ...

with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
    ensures{impl_post k phi result}
  = ...
```

The proofs of the nnfc and cnfc defunctionalized functions are similar to the proof of the impl_free function, being the code in Appendix A.7.

Results. The proof of correctness of the defunctionalized version of the T algorithm is naturally processed by Why3, with each proof objective being proved in less than one second as shown in Table 7.2.

Proof obligations	Alt-Ergo 2.2.0	CVC4 1.6
lemma VC for impl_free_defun	0.16	0.17
lemma VC for impl_apply	0.04	0.14
lemma VC for impl_defun_main	0.02	0.07
lemma VC for nnfc_defun	7.37	0.20
lemma VC for nnfc_apply	0.05	0.17
lemma VC for nnfc_defun_main	0.02	0.08
lemma VC for distr_defun	0.25	0.14
lemma VC for distr_apply	0.02	0.12
lemma VC for distr_defun_main	0.02	0.09
lemma VC for cnfc_defun	0.03	0.12
lemma VC for cnfc_apply	0.10	0.12
lemma VC for cnfc_defun_main	0.02	0.08
lemma VC for t	0.02	0.10

Table 7.2: Proof time of each defunctionalization proof obligation

7.3 Observations

The CPS and Defunctionalization techniques produces code with explicit stack structure, since each function call return a function (continuation). “Rewinding functions through CPS” contributes with a general solution to add trace and rewind functionalities to CPS programs [28].

The proofs were straightforward. However, Why3 operates on a first-order and the CPS transformation passes the program to a higher-order. In the Section 7.1 we showed how this can raise difficulties regarding its proof. The solution passes to have a version with an explicit stack structure but in a first-order language. This is obtained with the defunctionalization technique.

Chapter 8

Conclusions

The objective of this dissertation is to contribute to the development of pedagogical material to support Computational Logic courses, providing verified implementations of conversion algorithms to Conjunctive Normal Form and Horn Clauses. The focus of this work was that the implementation and verification effort should be adequate for undergraduate students.

The work presented shows that functional languages such as OCaml allow for implementations close to mathematical definitions without sacrificing clarity and rigour when presenting algorithms. This is adequate to be pedagogically used as an aid to the study and understanding of algorithms.

The proofs of these implementations are concise. For most functions, we ensure that the evaluation of the domain of our functions is equivalent to the evaluation of the codomain. Since we defined the theories before with all the properties needed, the proofs were clear and naturally processed. This reinforces the conclusion that it is feasible to show to students correctness proofs of the implementations.

In Chapter 7, given the importance of step-by-step executions when learning algorithms, we presented the CPS and Defunctionalization transformations of the conversion algorithm to CNF. These techniques permits the full control over program flow, allowing to, in the future, introduce a mechanism that can stop and resume the execution. The CPS transformation adds a function (continuation) as argument, thus turning it an higher-order function. However, in Why3 it is not possible to specify the nature of the continuation arguments, and Why3 also has limitations regarding the comparison of invariant types, which hampers the verification process. The defunctionalization technique produces a close version of a first-order abstract machine.

The transformations were manually made following the corresponding mechanical steps. It was straightforward but, in a more complex function, it can be a tedious process

(imagine, for example, applying the transformation to the `hornify_aux` function). It is more efficient to use tools and extensions to do it automatically. In the Section 3.3, we mentioned a extension developed also within the scope of the FACTOR project, that uses technologies to modify code and syntactic expressions to automatically transform functions into CPS. The idea is to run this extension for the transformation algorithm from CNF to Horn Clauses, as for further implementations.

This dissertation presents a successful proof of concept of formally verified bug-free implementations of (logical) algorithms. Every correct implementation and its verification will be incorporated in a library that will be available to students. Implementing and verifying algorithms increases their understanding and also slowly exposes students to formal methods. Developing fully bug-free programs is a dream for every programmer; we believe this a modest contribution towards it.

Future work. Notwithstanding the possibilities of expansion of this proof of concept, in the short term, we believe it is more important to:

1. Implement the Horn algorithm and perform its correction.
2. Implement the step-by-step execution.
3. Apply this approach to other algorithms of the Computational Logic courses, namely to the resolution algorithm.

However, in the long term, there are many paths this work can follow. For example, to apply this approach to other algorithms of the Computational Logic course, or to expand this approach to algorithms of other courses. It would be useful to create a web-tool with the step-by-step execution, that would allow students to run the algorithms with a certain input and follow each step of them. Moreover, the web-tool could, according to some implementation properties, ask students to specify each function, and then provide feedback on the correctness of the solutions submitted.

That's one small step for man, one giant leap for mankind.
Neil Armstrong

Bibliography

[1] W. Ackermann. "Frederic Brenton Fitch. Symbolic logic. An introduction." In: *Journal of Symbolic Logic* 17.4 (1952), 266–268. DOI: 10.2307/2266614.

[2] *Add injectivity for type invariant (#287) · Why3 Issues*. URL: https://gitlab.inria.fr/why3/why3/issues/287.

[3] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. "A Functional Correspondence Between Evaluators and Abstract Machines." In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. ACM, 2003. DOI: 10.1145/888251.888254.

[4] J. Aldrich. *Lecture Notes: Hoare Logic*. URL: https://www.cs.cmu.edu/~aldrich/courses/654-sp07/notes/3-hoare-notes.pdf.

[5] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. d. Sousa. *Rigorous Software Development: An Introduction to Program Verification*. 1st Edition. Undergraduate Topics in Computer Science. Springer, 2011. DOI: 10.1007/978-0-85729-018-2.

[6] C. Amaral, M. Florido, and V. Santos Costa. "PrologCheck – Property-Based Testing in Prolog." In: *Functional and Logic Programming*. Vol. 8475. Lecture Notes in Computer Science. Springer, 2014, pp. 1–17. DOI: 10.1007/978-3-319-07151-0_1.

[7] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[8] D. Barker-Plummer, J. Barwise, and J. Etchemendy. *Tarski's World: Revised and Expanded*. Center for the Study of Language and Information, 2007.

[9] D. Barker-Plummer, J. Barwise, and J. Etchemendy. *Language, Proof, and Logic: Second Edition*. 2nd Edition. Center for the Study of Language and Information, 2011.

[10] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. 1997.

[11] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. "CVC4." In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177.

[13] M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012. DOI: 10.1007/978-1-4471-4129-7.

[14] N. L. Biggs. *Matemática Discreta*. 1st Edition. Oxford University Press, 2002.

[15] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. *The Alt-Ergo automated theorem prover*. http://alt-ergo.lri.fr, 2008.

[16] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. "Why3: Shepherd Your Herd of Provers." In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64. URL: https://hal.inria.fr/hal-00790310.

[17] *Boole Manual*. URL: https://ggweb.gradegrinder.net/support/manual/boole.

[18] A. Bove, P. Dybjer, and U. Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types." In: *Theorem Proving in Higher Order Logics*. Vol. 5674. Lectures Notes in Computer Science. Springer, 2009, pp. 73–78.

[19] D. M. Cardoso. *Matemática Discreta*. 1st Edition. Escolar Editora, 2008.

[20] G. Chartrand and O. Oellermann. *Applied and Algorithmic Graph Theory*. International series in pure and applied mathematics. McGraw-Hill, 1993.

[21] R. Chen, C. Cohen, J. Lévy, S. Merz, and L. Théry. *Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle*. 2018. arXiv: 1810.11979.

[22] A. Cortesi, B. Le Charlier, and S. Rossi. "Specification-Based Automatic Verification of Prolog Programs." In: *Logic Program Synthesis and Transformation*. Vol. 1207. Lecture Notes in Computer Science. Springer, 1997, pp. 38–57. DOI: 10.1007/3-540-62718-9_3.

[23] A. CS2013. *Computer Science Curricula 2013. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Joint Task Force on Computing Curricula., 2013. URL: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

[24] D. van Dalen. *Logic and Structure*. Universitext. Springer, 2013.

[25] E. W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Commun. ACM* 18.8 (1975), pp. 453–457.

[26] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[27] H. Geuvers. "Proof assistants: History, ideas and future." In: *Sadhana* 34.1 (2009), pp. 3–25. DOI: 10.1007/s12046-009-0001-5.

[28] M. Giunti. *Rewinding functions through CPS*. 2019. URL: https://releaselab.gitlab.io/factor/pdfs/rewind_exp_report.pdf.

[29] E. G. Goodaire and M. M. Parmenter. *Discrete Mathematics with Graph Theory*. 1st Edition. Prentice Hall, 1997.

[30] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[31] L. Hamel. "Formal Methods : A First Introduction using Prolog to specify Programming Language Semantics." In: *Proceedings of the International Conference on Foundations of Computer Science*. 2016, pp. 70–76.

[32] A. G. Hamilton. *Logic for mathematicians*. Cambridge University Press, 1988.

[33] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 26 (1983), pp. 53–56. DOI: 10.1145/357980.358001.

[34] A. Horn. "On sentences which are true of direct unions of algebras." In: *Journal of Symbolic Logic* 16.1 (1951), 14–21. DOI: 10.2307/2268661.

[35] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[36] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems*. 2nd Edition. Cambridge University Press, 2004.

[37] *Informatics Degree - University of Algarve*. 2019. URL: https://www.ualg.pt/en/curso/1478.

[38] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java." In: *NASA Formal Methods*. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55.

[39] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by E. M. Clarke and A. Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.

[40] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.07: Documentation and user's manual*. Intern report. Inria, 2018. URL: https://hal.inria.fr/hal-00930213.

[41] F. Lindblad and M. Benke. "A Tool for Automated Theorem Proving in Agda." In: *Types for Proofs and Programs*. Vol. 3839. Lecture Notes in Computer Science. Springer, 2006, pp. 154–169. DOI: 10.1007/11617990_10.

[42] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. *Ariane 5 flight 501 failure report by the inquiry board*. European space agency Paris, 1996. URL: http://zoo.cs.yale.edu/classes/cs422/2010/bib/lions96ariane5.pdf.

[43] *Logic for Programming - Técnico Lisboa*. URL: https://fenix.tecnico.ulisboa.pt/cursos/leic-a/disciplina-curricular/1529008373638.

[44] *Lógica Computacional - Faculdade de Ciências da Universidade do Porto*. URL: https://sigarra.up.pt/fcup/pt/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=409431.

[45] *Lógica Computacional - FCT*. URL: http://lc.ssdi.di.fct.unl.pt/1819/web/index.html.

[46] *Lógica Computacional - UBI*. URL: http://www.di.ubi.pt/~desousa/LC/lc.html.

[47] *Lógica EI - Universidade do Minho*. URL: https://miei.di.uminho.pt/plano_estudos.html#l_gica_ei.

[48] J. P. Martins. *Lógica e Raciocínio*. 1st Edition. College Publications, 2014.

[49] *Matemática Discreta - Faculdade de Engenharia da Universidade do Porto*. URL: https://sigarra.up.pt/feup/pt/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=436426.

[50] *Matemática Discreta - Universidade de Aveiro*. URL: https://www.ua.pt/deti/uc/2585.

[51] E. Mendelson. *Introduction to mathematical logic*. 3rd Edition. Chapman and Hall, 1987.

[52] N. Moreira. *Lógica Computacional*. 2016. URL: http://www.dcc.fc.up.pt/~sandra//Home/LC1920_files/nlc.pdf.

[53] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[54] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016.

[55] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, et al. "Exploring the role of visualization and engagement in computer science education." In: *ACM Sigcse Bulletin*. Vol. 35. 2. ACM. 2002, pp. 131–152.

[56] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Lectures Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-45949-9.

[57] M. Pereira. *Desfuncionalizar para Provar*. 2019. arXiv: 1905.08368. URL: http://arxiv.org/abs/1905.08368.

[58] F. Pfenning. "Logical Frameworks—A Brief Introduction." In: *Proof and System-Reliability*. Vol. 62. NATO Science Series. Springer, 2002, pp. 137–166. DOI: 10.1007/978-94-010-0413-8_5.

[59] F. Pfenning and C. Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems." In: *Automated Deduction — CADE-16*. Vol. 1632. Lectures Notes in Computer Science. Springer, 1999, pp. 202–206.

[60] A. Ravara. "A Simple Functional Presentation and an Inductive Correctness Proof of the Horn Algorithm." In: *Electronic Proceedings in Theoretical Computer Science* 278 (Sept. 2018), pp. 34–48.

[61] A. Ravara. *Computational Logic: Objectives, syllabus, contents and teaching and assessment methods*. Private Communication, 2018.

[62] J. C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages." In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 363–397. DOI: 10.1023/A:1010027404223.

[63] T. Roxo, M. Pereira, and S. M. de Sousa. *Functional Programming with style and costless: CPS transformation "à la carte"*. 2019. URL: https://releaselab.gitlab.io/factor/pdfs/programacao_funcional_com_estilo.pdf.

[64] M. Sipser. *Introduction to the Theory of Computation*. 1st Edition. International Thomson Publishing, 1996.

[65] M. H. Stone. *The theory of representation for Boolean algebras*. Vol. 40. 1. JSTOR, 1936, pp. 37–111.

[66] G. J. Sussman and G. L. Steele. "Scheme: A Interpreter for Extended Lambda Calculus." In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 405–439. DOI: 10.1023/A:1010035624696.

[67] *Tezos Foundation*. URL: https://tezos.foundation/.

[68] *The Coq Proof Assistant*. URL: https://coq.inria.fr/about-coq.

[69] M Thomas and H Thimbleby. *Computer Bugs in Hospitals: An Unnoticed Killer*. 2018. URL: http://www.harold.thimbleby.net/killer.pdf.

[70] H. Trefftz. *Continuation–passing style in ML*. 2002.

[71] *Twelf Project*. URL: http://twelf.org/wiki/Main_Page.

[72] *Twelf Theorem Prover*. URL: https://www.cs.cmu.edu/~twelf/guide-1-4/twelf_10.html.

[73] F. Wiedijk. *The Seventeen Provers of the World*. Vol. 3600. 3600. Springer, 2006. DOI: 10.1007/11542384.

[74] A. Yushkovskiy. *Comparison of Two Theorem Provers: Isabelle/HOL and Coq*. 2018. arXiv: 1808.09701 [cs.LO].

Appendix 1 CNF Transformation Algorithm

A.1 Full Implementation

```
module T

  use booltheory.BoolImplementation, formula.LemmasAux, formula.
    PropositionalFormula, formula.ConjunctiveNormalForm, Size, int.Int

  let rec function impl_free (phi: formula) : formula_wi
  = match phi with
    | Prop t → if t = bot then L_wi (LBottom)
                else FNeg_wi (L_wi LBottom)
    | Var i →  L_wi (LVar i)
    | Neg phi1 → FNeg_wi (impl_free phi1)
    | Or phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
    | And phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
    | Impl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
    end

  let rec function nnfc (phi: formula_wi)
  = match phi with
    | FNeg_wi (FNeg_wi phi1) → nnfc phi1
    | FNeg_wi (FAnd_wi phi1 phi2) → FOr_nnf (nnfc (FNeg_wi phi1)) (nnfc (
    FNeg_wi phi2))
    | FNeg_wi (FOr_wi phi1 phi2) → FAnd_nnf (nnfc (FNeg_wi phi1)) (nnfc (
    FNeg_wi phi2))
    | FNeg_wi (L_wi phi1) → FNeg_nnf (phi1)
    | FOr_wi phi1 phi2 → FOr_nnf (nnfc phi1) (nnfc phi2)
    | FAnd_wi phi1 phi2 → FAnd_nnf (nnfc phi1) (nnfc phi2)
```

```
    | L_wi phi1 -> L_nnf phi1
  end

  let rec function distr (phi1 phi2: formula_cnf)
  = match phi1, phi2 with
    | FAnd_cnf phi11 phi12, phi2 -> FAnd_cnf (distr phi11 phi2) (distr phi12
    phi2)
    | phi1, FAnd_cnf phi21 phi22 -> FAnd_cnf (distr phi1 phi21) (distr phi1
    phi22)
    | FClause_cnf phi1, FClause_cnf phi2 -> FClause_cnf (DOr_cnf phi1 phi2)

    end

  let rec function cnfc (phi: formula_nnf)
  = match phi with
    | FOr_nnf phi1 phi2 -> distr (cnfc phi1) (cnfc phi2)
    | FAnd_nnf phi1 phi2 -> FAnd_cnf (cnfc phi1) (cnfc phi2)
    | FNeg_nnf literal -> FClause_cnf (DNeg_cnf literal)
    | L_nnf literal -> FClause_cnf (DLiteral literal)
  end

  let t (phi: formula) : formula_cnf
  = cnfc (nnfc (impl_free phi))

end
```

A.2 Evaluation Functions

```
  function assign (e : formula) (f : i -> t)  : formula =
    match e with
    | Prop t -> Prop t
    | Var i -> Prop (f i)
    | Neg e -> Neg (assign e f)
    | And e1 e2 -> And (assign e1 f) (assign e2 f)
    | Or e1 e2 -> Or (assign e1 f) (assign e2 f)
    | Impl e1 e2 -> Impl (assign e1 f) (assign e2 f)
    end

  function eval_recursive (e : formula) : t =
    match e with
    | Prop t -> t
    | Neg e1 -> neg (eval_recursive e1)
    | And e1 e2 -> (eval_recursive e1) /*\ (eval_recursive e2)
    | Or e1 e2 -> (eval_recursive e1) \*/ (eval_recursive e2)
    | Impl e1 e2 -> (eval_recursive e1) ->* (eval_recursive e2)
```

```
  | _ → bot (* never reached *)
  end

function eval (e : formula) (f : i → t) : t =
  eval_recursive (assign e f)

function eval_pliteral (l: pliteral) (f: i → t) : t
= match l with
    | LBottom → bot
    | LVar i → f i
  end

function eval_wi (fwi: formula_wi) (f: i → t) : t
= match fwi with
    | L_wi phi1 → eval_pliteral phi1 f
    | FAnd_wi fwi1 fwi2  → eval_wi fwi1 f /*\ eval_wi fwi2 f
    | FOr_wi fwi1 fwi2   → eval_wi fwi1 f \*/ eval_wi fwi2 f
    | FNeg_wi fwi      → neg (eval_wi fwi f)
  end

function eval_nnf (fnnf: formula_nnf) (f: i → t)  : t
= match fnnf with
    | FNeg_nnf literal → neg (eval_pliteral literal f)
    | L_nnf literal → eval_pliteral literal f
    | FAnd_nnf fnnf1 fnnf2 → eval_nnf fnnf1 f /*\ eval_nnf fnnf2 f
    | FOr_nnf fnnf1 fnnf2 → eval_nnf fnnf1 f \*/ eval_nnf fnnf2 f
  end

function eval_clause_cnf (fcnf: clause_cnf) (f: i → t) : t
= match fcnf with
    | DLiteral l → eval_pliteral l f
    | DNeg_cnf l → neg (eval_pliteral l f)
    | DOr_cnf phi1 phi2 → eval_clause_cnf phi1 f \*/ eval_clause_cnf phi2 f
  end

function eval_formula_cnf (e: formula_cnf) (f: i → t) : t
= match e with
    | FClause_cnf phi1 → eval_clause_cnf phi1 f
    | FAnd_cnf phi1 phi2 → eval_formula_cnf phi1 f /*\ eval_formula_cnf phi2
  f
  end
```

A.3 Direct Style Proof

```
let rec function impl_free (phi: formula) : formula_wi
  variant{ phi }
  ensures{ forall f. eval phi f = eval_wi result f }
= ...

let rec function nnfc (phi: formula_wi)
  variant{ size phi }
  ensures{ (forall f. eval_wi phi f = eval_nnf result f)}
= ...

let rec function distr (phi1 phi2: formula_cnf)
  variant{ size_cnf phi1 + size_cnf phi2 }
  ensures{ (forall f. ((eval_formula_cnf phi1 f \*/ eval_formula_cnf phi2 f) =
   eval_formula_cnf result f)) }
= ...

let rec function cnfc (phi: formula_nnf)
  variant{ phi }
  ensures{ (forall f. eval_nnf phi f = eval_formula_cnf result f) }
= ...

let t (phi: formula) : formula_cnf
  ensures{ (forall f. eval phi f = eval_formula_cnf result f)}
= ...
```

A.4 CPS Version

```
module T_CPS

  use booltheory.BoolImplementation, formula.PropositionalFormula, formula.
    ConjunctiveNormalForm, T, Size, int.Int

  let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  = match phi with
    | Prop t → if t = bot then k (L_wi (LBottom))
                else k (FNeg_wi (L_wi LBottom))
    | Var i → k (L_wi (LVar i))
    | Neg phi1 → impl_free_cps phi1 (fun con → k (FNeg_wi con))
    | Or phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (fun
    con1 → k (FOr_wi con con1)))
    | And phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (fun
    con1 → k (FAnd_wi con con1)))
    | Impl phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (fun
    con1 → k (FOr_wi (FNeg_wi con) con1)))
    end

  let impl_free_main (phi: formula) : formula_wi
    ensures{forall f. eval phi f = eval_wi result f}
  = impl_free_cps phi (fun x → x)

  let rec nnfc_cps (phi: formula_wi) (k: formula_nnf → 'a) : 'a
  = match phi with
    | FNeg_wi (FNeg_wi phi1) → nnfc_cps phi1 (fun con → k con)
    | FNeg_wi (FAnd_wi phi1 phi2) → nnfc_cps (FNeg_wi phi1) (fun con →
    nnfc_cps (FNeg_wi phi2) (fun con1 → k (FOr_nnf con con1)))
    | FNeg_wi (FOr_wi phi1 phi2) → nnfc_cps (FNeg_wi phi1) (fun con → nnfc_cps
     (FNeg_wi phi2) (fun con1 → k (FAnd_nnf con con1)))
    | FOr_wi phi1 phi2 → nnfc_cps phi1 (fun con → nnfc_cps phi2 (fun con1 → k
     (FOr_nnf con con1)))
    | FAnd_wi phi1 phi2 → nnfc_cps phi1 (fun con → nnfc_cps phi2 (fun con1 →
    k (FAnd_nnf con con1)))
    | FNeg_wi (L_wi phi1) → k (FNeg_nnf phi1)
    | L_wi phi1 → k (L_nnf phi1)
  end

  let nnfc_main (phi: formula_wi) : formula_nnf
    ensures{(forall f. eval_wi phi f = eval_nnf result f)}
  = nnfc_cps phi (fun x → x)

  let rec distr_cps (phi1 phi2: formula_cnf) (k: formula_cnf → 'a) : 'a
  = match phi1, phi2 with
```

```
    | FAnd_cnf phi11 phi12, phi2 → distr_cps phi11 phi2 (fun con → distr_cps
    phi12 phi2 (fun con1 → k (FAnd_cnf con con1)))
    | phi1, FAnd_cnf phi21 phi22 → distr_cps phi1 phi21 (fun con → distr_cps
    phi1 phi22 (fun con1 → k (FAnd_cnf con con1)))
    | FClause_cnf phi1, FClause_cnf phi2 → k (FClause_cnf (DOr_cnf phi1 phi2))
    end

  let distr_main (phi1 phi2: formula_cnf) : formula_cnf
  = distr_cps phi1 phi2 (fun x → x)


  let rec cnfc_cps (phi: formula_nnf) (k: formula_cnf → 'a) : 'a
  = match phi with
    | FOr_nnf phi1 phi2 → cnfc_cps phi1 (fun con → cnfc_cps phi2 (fun con1 →
    distr_cps con con1 k))
    | FAnd_nnf phi1 phi2 → cnfc_cps phi1 (fun con → cnfc_cps phi2 (fun con1 →
     k (FAnd_cnf con con1)))
    | FNeg_nnf literal → k (FClause_cnf (DNeg_cnf literal))
    | L_nnf literal → k (FClause_cnf (DLiteral literal))
  end

  let cnfc_main (phi: formula_nnf) : formula_cnf
  = cnfc_cps phi (fun x → x)

  let t_main (phi: formula) : formula_cnf
  = cnfc_cps (nnfc_cps (impl_free_cps (phi) (fun x → x)) (fun x → x)) (fun x
    → x)

end
```

A.5 CPS Proof

```
module T_CPS

  use booltheory.BoolImplementation, formula.PropositionalFormula, formula.
    ConjunctiveNormalForm, T, Size, int.Int

  let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
    variant{ phi }
    ensures{ result = k (impl_free phi) }
  = ...

  let impl_free_main (phi: formula) : formula_wi
    ensures{forall f. eval phi f = eval_wi result f}
  = ...
```

```
  let rec nnfc_cps (phi: formula_wi) (k: formula_nnf → 'a) : 'a
    variant{ size phi }
    ensures{ result = k (nnfc phi) }
  = ...

  let nnfc_main (phi: formula_wi) : formula_nnf
    ensures{(forall f. eval_wi phi f = eval_nnf result f)}
  = ...

  let rec distr_cps (phi1 phi2: formula_cnf) (k: formula_cnf → 'a) : 'a
    variant{ size_cnf phi1 + size_cnf phi2 }
    ensures{ result = k (distr phi1 phi2) }
  = ...

  let distr_main (phi1 phi2: formula_cnf) : formula_cnf
    ensures { (forall f. ((eval_formula_cnf phi1 f \*/ eval_formula_cnf phi2 f)
    = eval_formula_cnf result f)) }
  = ...


  let rec cnfc_cps (phi: formula_nnf) (k: formula_cnf → 'a) : 'a
    variant{ phi }
    ensures{ result = k (cnfc phi)}
  = ...

  let cnfc_main (phi: formula_nnf) : formula_cnf
    ensures{ (forall f. eval_nnf phi f = eval_formula_cnf result f) }
  = ...

  let t_main (phi: formula) : formula_cnf
    ensures{ (forall f. eval phi f = eval_formula_cnf result f) }
  = ...

end
```

A.6 Defunctionalized Version

```
module Defunctionalization

  use booltheory.BoolImplementation, formula.PropositionalFormula, formula.
    ConjunctiveNormalForm, T, Size, int.Int

  (* TYPES *)

  type impl_kont =
    | KImpl_Id
    | KImpl_Neg impl_kont formula
    | KImpl_OrLeft formula impl_kont
    | KImpl_OrRight impl_kont formula_wi
    | KImpl_AndLeft formula impl_kont
    | KImpl_AndRight impl_kont formula_wi
    | KImpl_ImplLeft formula impl_kont
    | KImpl_ImplRight impl_kont formula_wi

  type nnfc_kont =
    | Knnfc_id
    | Knnfc_negneg nnfc_kont formula_wi
    | Knnfc_negandleft formula_wi nnfc_kont
    | Knnfc_negandright nnfc_kont formula_nnf
    | Knnfc_negorleft formula_wi nnfc_kont
    | Knnfc_negorright nnfc_kont formula_nnf
    | Knnfc_andleft formula_wi nnfc_kont
    | Knnfc_andright nnfc_kont formula_nnf
    | Knnfc_orleft formula_wi nnfc_kont
    | Knnfc_orright nnfc_kont formula_nnf

  type distr_kont =
    | KDistr_Id
    | KDistr_Left formula_cnf formula_cnf distr_kont
    | KDistr_Right distr_kont formula_cnf


  type cnfc_kont =
    | KCnfc_Id
    | KCnfc_OrLeft formula_nnf cnfc_kont
    | KCnfc_OrRight cnfc_kont formula_cnf
    | KCnfc_AndLeft formula_nnf cnfc_kont
    | KCnfc_AndRight cnfc_kont formula_cnf


  (* DESF FUNCTIONS *)
```

```
(* IMPL_FREE *)

let rec impl_free_defu (phi: formula) (k: impl_kont ) : formula_wi
= match phi with
  | Prop t → if t = bot then impl_apply (L_wi (LBottom)) k
             else impl_apply (FNeg_wi (L_wi LBottom)) k
  | Var i → impl_apply (L_wi (LVar i)) k
  | Neg phi1 → impl_free_defu phi1 (KImpl_Neg k phi1)
  | Or phi1 phi2 → impl_free_defu phi1 (KImpl_OrLeft phi2 k)
  | And phi1 phi2 → impl_free_defu phi1 (KImpl_AndLeft phi2 k)
  | Impl phi1 phi2 → impl_free_defu phi1 (KImpl_ImplLeft phi2 k)
  end

with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
= match k with
  | KImpl_Id → phi
  | KImpl_Neg k phi1 → impl_apply (FNeg_wi phi) k
  | KImpl_OrLeft phi1 k → impl_free_defu phi1 (KImpl_OrRight k phi)
  | KImpl_OrRight k phi2 → impl_apply (FOr_wi phi2 phi) k
  | KImpl_AndLeft phi1 k → impl_free_defu phi1 (KImpl_AndRight k phi)
  | KImpl_AndRight k phi2 → impl_apply (FAnd_wi phi2 phi) k
  | KImpl_ImplLeft phi1 k → impl_free_defu phi1 (KImpl_ImplRight k phi)
  | KImpl_ImplRight k phi2→ impl_apply (FOr_wi (FNeg_wi phi2) phi) k
end

let rec impl_defu_main (phi:formula) : formula_wi
= impl_free_defu phi KImpl_Id


(* NNFC *)

let rec nnfc_defu (phi: formula_wi) (k: nnfc_kont) : formula_nnf
= match phi with
  | FNeg_wi (FNeg_wi phi1) → nnfc_defu phi1 (Knnfc_negneg k phi1)
  | FNeg_wi (FAnd_wi phi1 phi2) → nnfc_defu (FNeg_wi phi1) (Knnfc_negandleft
  phi2 k)
  | FNeg_wi (FOr_wi phi1 phi2) → nnfc_defu (FNeg_wi phi1) (Knnfc_negorleft
  phi2 k)
  | FOr_wi phi1 phi2 → nnfc_defu phi1 (Knnfc_orleft phi2 k)
  | FAnd_wi phi1 phi2 → nnfc_defu phi1 (Knnfc_andleft phi2 k)
  | FNeg_wi (L_wi phi1) → nnfc_apply (FNeg_nnf phi1) k
  | L_wi phi1 → nnfc_apply (L_nnf phi1) k
end
```

```
with nnfc_apply (phi: formula_nnf) (k: nnfc_kont) : formula_nnf
= match k with
  | Knnfc_id → phi
  | Knnfc_negneg k phi1 → nnfc_apply phi k
  | Knnfc_negandleft phi1 k → nnfc_defu (FNeg_wi phi1) (Knnfc_negandright k
  phi)
  | Knnfc_negandright k phi2 → nnfc_apply (FOr_nnf phi2 phi) k
  | Knnfc_negorleft phi1 k → nnfc_defu (FNeg_wi phi1) (Knnfc_negorright k phi
  )
  | Knnfc_negorright k phi2 → nnfc_apply (FAnd_nnf phi2 phi) k
  | Knnfc_andleft phi1 k → nnfc_defu phi1 (Knnfc_andright k phi)
  | Knnfc_andright k phi2 → nnfc_apply (FAnd_nnf phi2 phi) k
  | Knnfc_orleft phi1 k → nnfc_defu phi1 (Knnfc_orright k phi)
  | Knnfc_orright k phi2 → nnfc_apply (FOr_nnf phi2 phi) k
end

let nnfc_defu_main (phi: formula_wi) : formula_nnf
= nnfc_defu phi Knnfc_id

(* Distr *)

let rec distr_defu (phi1 phi2: formula_cnf) (k: distr_kont) : formula_cnf
= match phi1, phi2 with
  | FAnd_cnf phi11 phi12, phi2 → distr_defu phi11 phi2 (KDistr_Left phi12
  phi2 k)
  | phi1, FAnd_cnf phi21 phi22 → distr_defu phi1 phi21 (KDistr_Left phi1
  phi22 k)
  | FClause_cnf phi1, FClause_cnf phi2 → distr_apply (FClause_cnf (DOr_cnf
  phi1 phi2)) k
  end

with distr_apply (phi: formula_cnf) (k: distr_kont) : formula_cnf
= match k with
  | KDistr_Id → phi
  | KDistr_Left phi1 phi2 k →
      distr_defu phi1 phi2 (KDistr_Right k phi)
  | KDistr_Right k phi1 →
      distr_apply (FAnd_cnf phi1 phi) k
end

let distr_defu_main (phi1 phi2: formula_cnf) : formula_cnf
= distr_defu phi1 phi2 KDistr_Id

(* CNFC *)

let rec cnfc_defu (phi: formula_nnf) (k: cnfc_kont) : formula_cnf
```

```
  = match phi with
    | FOr_nnf phi1 phi2 → cnfc_defu phi1 (KCnfc_OrLeft phi2 k)
    | FAnd_nnf phi1 phi2 → cnfc_defu phi1 (KCnfc_AndLeft phi2 k)
    | FNeg_nnf literal → cnfc_apply (FClause_cnf (DNeg_cnf literal)) k
    | L_nnf literal → cnfc_apply (FClause_cnf (DLiteral literal)) k
  end

  with cnfc_apply (phi: formula_cnf) (k: cnfc_kont) : formula_cnf
  = match k with
    | KCnfc_Id → phi
    | KCnfc_OrLeft phi1 k →
        cnfc_defu phi1 (KCnfc_OrRight k phi)
    | KCnfc_OrRight k phi2 →
        cnfc_apply (distr_defu phi2 phi KDistr_Id) k
    | KCnfc_AndLeft phi1 k →
        cnfc_defu phi1 (KCnfc_AndRight k phi)
    | KCnfc_AndRight k phi2 →
        cnfc_apply (FAnd_cnf phi2 phi) k
  end

  let cnfc_defu_main (phi: formula_nnf) : formula_cnf
  = cnfc_defu phi KCnfc_Id

  let t (phi: formula) : formula_cnf
  = cnfc_defu_main ( nnfc_defu_main ( impl_defu_main phi))

end
```

A.7 Defunctionalized Proof

```
(* POSTS *)

predicate impl_post (k: impl_kont) (phi result: formula_wi)
= match k with
  | KImpl_Id → let x = phi in x = result
  | KImpl_Neg k phi1 → let neg = phi in impl_post k (FNeg_wi phi) result
  | KImpl_OrLeft phi1 k → let hl = phi in impl_post k (FOr_wi phi (impl_free
    phi1)) result
  | KImpl_OrRight k phi2 → let hr = phi in impl_post k (FOr_wi phi2 hr) result
  | KImpl_AndLeft phi1 k → let hl = phi in impl_post k (FAnd_wi phi (impl_free
    phi1)) result
  | KImpl_AndRight k phi2 → let hr = phi in impl_post k (FAnd_wi phi2 hr)
    result
  | KImpl_ImplLeft phi1 k → let hl = phi in impl_post k (FOr_wi (FNeg_wi phi) (
    impl_free phi1)) result
```

```
  | KImpl_ImplRight k phi2→ let hr = phi in impl_post k (FOr_wi (FNeg_wi phi2)
    hr) result
end

predicate nnfc_post (k: nnfc_kont) (phi result: formula_nnf)
= match k with
  | Knnfc_id → let x = phi in x = result
  | Knnfc_negneg k phi1 → let neg = phi in nnfc_post k phi result
  | Knnfc_negandleft phi1 k → let hl = phi in nnfc_post k (FOr_nnf phi (nnfc (
    FNeg_wi phi1))) result
  | Knnfc_negandright k phi2 → let hr = phi in nnfc_post k (FOr_nnf phi2 hr)
    result
  | Knnfc_negorleft phi1 k → let hl = phi in nnfc_post k (FAnd_nnf phi (nnfc (
    FNeg_wi phi1))) result
  | Knnfc_negorright k phi2 →let hr = phi in nnfc_post k (FAnd_nnf phi2 hr)
    result
  | Knnfc_andleft phi1 k → let hl = phi in nnfc_post k (FAnd_nnf phi (nnfc phi1
    )) result
  | Knnfc_andright k phi2 → let hr = phi in nnfc_post k (FAnd_nnf phi2 hr)
    result
  | Knnfc_orleft phi1 k → let hl = phi in nnfc_post k (FOr_nnf phi (nnfc phi1))
     result
  | Knnfc_orright k phi2 → let hr = phi in nnfc_post k (FOr_nnf phi2 hr) result
end

predicate distr_post (k: distr_kont) (phi result: formula_cnf)
= match k with
  | KDistr_Id → let x = phi in x = result
  | KDistr_Left phi1 phi2 k → let hl = phi in distr_post k (FAnd_cnf hl (distr
    phi1 phi2)) result
  | KDistr_Right k phi1 → let hr = phi in distr_post k (FAnd_cnf phi1 hr)
    result
end

predicate cnfc_post (k: cnfc_kont) (phi result: formula_cnf)
= match k with
  | KCnfc_Id → let x = phi in x = result
  | KCnfc_OrLeft phi1 k → let hl = phi in cnfc_post k (distr hl (cnfc phi1))
    result
  | KCnfc_OrRight k phi2 → let hr = phi in cnfc_post k (distr phi2 hr) result
  | KCnfc_AndLeft phi1 k → let hl = phi in cnfc_post k (FAnd_cnf phi (cnfc phi1
    )) result
  | KCnfc_AndRight k phi2 → let hr = phi in cnfc_post k (FAnd_cnf phi2 hr)
    result
end
```

```
(* FUNCTIONS *)

let rec impl_free_defu (phi: formula) (k: impl_kont ) : formula_wi
  diverges
  ensures{ impl_post k (impl_free phi) result }
= ...

with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
  diverges
  ensures{impl_post k phi result}
= ...

let rec impl_defu_main (phi:formula) : formula_wi
  diverges
  ensures{ forall f. eval phi f = eval_wi result f }
= ...

let rec nnfc_defu (phi: formula_wi) (k: nnfc_kont) : formula_nnf
  diverges
  ensures{ nnfc_post k (nnfc phi) result }
= ...

with nnfc_apply (phi: formula_nnf) (k: nnfc_kont) : formula_nnf
  diverges
  ensures{ nnfc_post k phi result }
= ...

let nnfc_defu_main (phi: formula_wi) : formula_nnf
  diverges
  ensures { forall f. eval_wi phi f = eval_nnf result f }
= ...

let rec distr_defu (phi1 phi2: formula_cnf) (k: distr_kont) : formula_cnf
  diverges
  ensures{ distr_post k (distr phi1 phi2) result }
= ...

with distr_apply (phi: formula_cnf) (k: distr_kont) : formula_cnf
  diverges
  ensures{ distr_post k phi result }
= ...

let distr_defu_main (phi1 phi2: formula_cnf) : formula_cnf
  diverges
  ensures { forall f. ((eval_formula_cnf phi1 f \*/ eval_formula_cnf phi2 f) =
    eval_formula_cnf result f) }
```

```
= ...

let rec cnfc_defu (phi: formula_nnf) (k: cnfc_kont) : formula_cnf
  diverges
  ensures{ cnfc_post k (cnfc phi) result }
= ...

with cnfc_apply (phi: formula_cnf) (k: cnfc_kont) : formula_cnf
  diverges
  ensures{ cnfc_post k phi result }
= ...fc_apply (FAnd_cnf phi2 phi) k
end

let cnfc_defu_main (phi: formula_nnf) : formula_cnf
  diverges
  ensures{ forall f. eval_nnf phi f = eval_formula_cnf result f }
= ...

let t (phi: formula) : formula_cnf
  diverges
  ensures{  forall f. eval phi f = eval_formula_cnf result f }
= ...
```

Appendix B

Appendix 2 Hornify

B.1 Full Implementation

```
module Hornify

  use import formula.ConjunctiveNormalForm as CNF
  use import formula.Horn as Horn

  use booltheory.BoolImplementation, int.Int, setstheory.BoolSet, setstheory.
    PropositionalFormulaSet, option.Option

  clone export set.SetApp with type elt = pliteral

  exception MoreThanOnePositive

  let convertLiteralToR (pl: pliteral) : (rightside)
  = match pl with
    | LBottom → RProp bot
    | LVar x → RVar x
    end

  let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside)
  : (rs: set, rp: option rightside)
  = match pl with
    | LBottom → let rbottom = Some (RProp bot) in
                match nl with
                | LBottom → ((add (Prop_pl bot) s), rbottom)
                | LVar x → ((add (Var_pl x) s), rbottom)
                end
```

```
  | LVar x → let rvar = Some (RVar x) in
              match nl with
              | LBottom → ((add (Prop_pl bot) s), rvar)
              | LVar x → ((add (Var_pl x) s), rvar)
              end
  end

let processCombination (pl: pliteral) (nl: pliteral) (s: set)
(p: option rightside) : (rs: set, rp: option rightside)
raises{ MoreThanOnePositive }
= match p with
  | None → addLiterals pl nl s p
  | Some _ → raise MoreThanOnePositive
  end

let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside)
: (rs: set, rp: option rightside)
raises{ MoreThanOnePositive }
= match phi with
  | DOr_cnf (DLiteral _) (DLiteral _) → raise MoreThanOnePositive
  | DOr_cnf (DLiteral pl) (DNeg_cnf nl)
  | DOr_cnf (DNeg_cnf nl) (DLiteral pl) → processCombination pl nl s p
  | DOr_cnf (DNeg_cnf nl1) (DNeg_cnf nl2) →
      ((add (convertPLiteralToPL nl1) (add (convertPLiteralToPL nl2) s)), p)
  | DOr_cnf (DOr_cnf phi1 phi2) (DLiteral pl)
  | DOr_cnf (DLiteral pl) (DOr_cnf phi1 phi2) →
      match p with
        | None →
            hornify_aux (DOr_cnf phi1 phi2) s (Some (convertLiteralToR pl))
        | Some _ → raise MoreThanOnePositive
      end
  | DOr_cnf (DOr_cnf phi1 phi2) (DNeg_cnf nl)
  | DOr_cnf (DNeg_cnf nl) (DOr_cnf phi1 phi2) →
       hornify_aux (DOr_cnf phi1 phi2) (add (convertPLiteralToPL nl) s) p
  | DOr_cnf phi1 phi2 →
      let (s1,p1) = hornify_aux phi1 s p in hornify_aux phi2 s1 p1
  | _ → absurd
  end

let buildConjunction (s: set): leftside
= let rec build (s: set)
   = if(is_empty s) then absurd else
     if((cardinal s) = 1) then (convertPLtoPLC (choose s)) else
       let element = choose s in
        PLCAnd (convertPLtoPLC (element)) (build (remove (element) s)) in
  LPos (build s)
```

```
  let getPositive (p: option rightside) : rightside
  = match p with
      | None → RProp bot
      | Some x → x
    end

  let getBasicHorn (phi: clause_cnf) : basic_horn_clause
  = match phi with
    | DLiteral (LVar x) → BImpl (LTop) (RVar x)
    | DLiteral (LBottom) → BImpl (LTop) (RProp bot)
    | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RProp bot)
    | DNeg_cnf (LBottom) → BImpl (LTop) (RProp top)
    | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
                          BImpl (buildConjunction s) (getPositive p)
    end

  let rec hornify (phi: formula_cnf) : hornclause
  = match phi with
    | FClause_cnf phi1 → HBasic (getBasicHorn phi1)
    | FAnd_cnf phi1 phi2 → HAnd (hornify phi1) (hornify phi2)
    end

end
```

B.2 Evaluation Functions

```
module Valuation

  use booltheory.BoolImplementation, ConjunctiveNormalForm, option.Option

  function assign_rightside (r: rightside) (f: i → t) : rightside
  = match r with
    | RProp t → RProp t
    | RVar i → RProp (f i)
    end

  function eval_rightside (r: rightside) : t
  = match r with
    | RProp t → t
    | _ → bot
    end
```

```
  function eval_positive (plc: conj_pliteral) (f: i → t) : t
  = match plc with
    | CPL l → eval_pliteral l f
    | CPAnd phi1 phi2 → eval_positive phi1 f /*\ (eval_positive phi2 f)
    end

  function eval_leftside (l: leftside) (f: i → t) : t
  = match l with
    | LTop → top
    | LPos phi1 → eval_positive phi1 f
    end

  function eval_basichornclause (b: basichornclause) (f: i → t) : t
  = match b with
    | BImpl left right → (eval_leftside left f) →* (eval_rightside (
    assign_rightside right f))
    end

  function eval_hornclause (h: hornclause) (f: i → t) : t
  = match h with
    | HBasic h1 → eval_basichornclause h1 f
    | HAnd h1 h2 → eval_hornclause h1 f /*\ eval_hornclause h2 f
    end

  function eval_optionrightside (p: option rightside) (f: i → t) : t
  = match p with
    | None → bot
    | Some x → eval_rightside (assign_rightside x f)
    end

end
```

B.3 Hornify Proof

```
module Hornify

  ...

  let convertLiteralToR (pl: pliteral) : (rightside)
    ensures{ forall f. CNF.eval_pliteral pl f = Horn.eval_rightside (
    assign_rightside result f) }
  = ...
```

```
let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) :
   (rs: set, rp: option rightside)
  requires{ p = None }
  ensures{ (not is_empty rs) }
  ensures { forall f. (eval_pliteral pl f \*/ (neg (eval_pliteral nl f)) \*/
  eval_negative s f \*/ eval_optionrightside p f) = (eval_negative rs f \*/
  eval_optionrightside rp f) }
= ...

let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option
  rightside) : (rs: set, rp: option rightside)
  raises{ MoreThanOnePositive }
  ensures{ (not is_empty rs) }
  ensures { forall f. (eval_pliteral pl f \*/ (neg (eval_pliteral nl f)) \*/
  eval_negative s f \*/ eval_optionrightside p f) = (eval_negative rs f \*/
  eval_optionrightside rp f) }
= ...

let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs:
  set, rp: option rightside)
  requires{ exists x y. phi = DOr_cnf x y }
  ensures{ (not is_empty rs) }
  ensures{ forall f. eval_domain phi s p f = eval_codomain rs rp f }
  ensures{ forall f. eval_domain phi s p f = ((eval_positive rs f) →* (
  eval_optionrightside rp f)) }
  raises{ MoreThanOnePositive }
  variant{ phi }
= ...

let conjunction (s: set): leftside
  requires{not is_empty s}
  ensures{forall f. eval_positive s f = eval_leftside result f }
= ...

let getPositive (p: option rightside) : rightside
  ensures{forall f. eval_optionrightside p f = eval_rightside (
  assign_rightside result f)}
= ...

let getBasicHorn (phi: clause_cnf) : basichornclause
  ensures{ forall f. eval_clause_cnf phi f = eval_basichornclause result f }
  raises{ MoreThanOnePositive }
= ...
```

```
  let rec hornify (phi: formula_cnf) : hornclause
    raises{ MoreThanOnePositive }
    ensures{forall f. eval_formula_cnf phi f = eval_hornclause result f}
    variant{ phi }
  = ...

end
```