Pedro Miguel Laforêt Barroso

Bachelor Degree

Formally Verified Bug-free Implementations of (Logical) Algorithms

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: António Ravara,
Associate Professor,
Faculdade de Ciências e Tecnologia da Universidade
Nova de Lisboa

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Formally Verified Bug-free Implementations of (Logical) Algorithms

Copyright © Pedro Miguel Laforêt Barroso, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da 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

This dissertation was supported by the Tezos Foundation.

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

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

Computational Logic is an essential field of Computer Science. Courses on this subject are either too informal (only providing pseudo-codes) or too formal when describing algorithms. In either case, there is an emphasis on paper-and-pencil proofs rather than on computational approaches. It is seldom the case that there are running algorithms (executable programs). Implementations of algorithms and tools that help students go through the resolutions of exercises are crucial, as they are an important pedagogical object.

Most courses make only a small or no effort on showing the presented algorithms correction, although this is an important aspect of software development that should be natural for Computer Science students. For instance, the Integrated Master in Computer Science course of the Faculty of Science and Technology of NOVA University of Lisbon only starts referring to program verification in the 4th year. Students often make wrong assumptions over the correction of their programs basing themselves in tests, which are not enough to prove that a program is completely bug-free. Tests only discover the presence of bugs. To assure that a program is correct we must use formal verification instead.

The purpose of this dissertation is to contribute to the formalization and assisted-verification of standard and classical algorithms of Computational Logic, through the development of step-by-step implementations of those algorithms in a functional language (the nature of these languages offers a closer relationship to mathematical definitions) and posterior verification of such implementations using the Why3 program verification platform. These implementations and soundness proofs will serve as pedagogical material to Computational Logic students.

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

Resumo

Lógica Computacional é um campo essencial para Engenharia Informática. Cursos sobre este assunto são muito informais (apenas fornecendo pseudo-códigos) ou muito formais ao descrever algoritmos. Em ambos os casos, existe um empatia por provas a papel e caneta em vez de abordagens computacionais. É raro o caso onde existem algoritmos em execução. Implementações e ferramentas que ajudem os alunos a analisarem as resoluções dos exercícios são cruciais, pois são um importante objeto pedagógico.

A maioria dos cursos faz apenas um pequeno ou nenhum esforço em mostrar a correção de algoritmos apresentados, embora seja um aspeto importante no desenvolvimento de software que deve ser natural para estudantes de Informática. Por exemplo, o curso de Mestrado Integrado em Engenharia Informática da Universidade Nova de Lisboa só começa a referir verificação de programa a partir do 4º ano. Os estudantes muitas vezes fazem suposições erradas sobre a correção dos seus programas, baseando-se em testes, coisa que não é suficiente para provar que um programa é completamente livre de bugs. Testes só descobrem a presença de bugs, portanto para garantir que um programa está correto e livre de bugs, devemos usar a verificação formal.

O objetivo desta dissertação é contribuir para a formalização e verificação assistida de algoritmos clássicos de Lógica Computacional, através do desenvolvimento de implementações passo-a-passo destes mesmos algoritmos numa linguagem funcional (a natureza deste tipo de linguagens oferece uma relação mais próxima com as definições matemáticas) e posterior verificação destas implementações usando a ferramenta de verificação de programas, Why3. Estas implementações e provas de correção servirão de material pedagógico para estudantes de lógica computacional.

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

Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 FACTOR . . . 2
1.3 Problem . . . 2
1.4 Objective . . . 3
1.5 Contributions . . . 4
1.6 Document Structure . . . 4

2 State of the Art **5**

2.1 Current State of Computational Logic in Portugal . . . 5
2.2 Tools to support Computational Logic . . . 7
2.2.1 Tarski's World . . . 7
2.2.2 Fitch . . . 8
2.2.3 Boole . . . 9
2.3 Proof-Assistant Tools . . . 10
2.3.1 Why3 / Dafny . . . 10
2.3.2 Coq / Isabelle / Agda . . . 10
2.3.3 CFML . . . 10
2.3.4 Twelf . . . 11
2.3.5 Conclusions . . . 11

3 Background and Preliminaries **13**

3.1 Program Verification . . . 13
3.2 Why3 . . . 15
3.3 Continuation-Passing Style . . . 16
3.4 Defuncionalization . . . 17

4 Conjunctive Normal Form Transformation Algorithm **19**

4.1 Functional presentation of the algorithm . . . 19
4.2 Implementation . . . 20
4.3 How to obtain the correctness . . . 23
4.4 Proof of correctness . . . 24
4.5 Continuation-Passing Style . . . 25

4.6 Conclusions and Observations 27
4.6.1 Previous implementation and proof of correctness. 27
4.6.2 Defunctionalization 30

5 Hornify **35**
5.1 Algorithm Definition 35
5.2 Functional Presentation of the Algorithm 36
5.3 Implementation 37
5.4 Verification 40
5.5 Conclusions and Observations 42

6 Conclusions **43**

Bibliography **45**

A Appendix 1 CNF Transformation Algorithm **49**
A.1 Full Implementation 49
A.2 Evaluation Functions 51
A.3 Direct Style Proof 52
A.4 CPS Version 54
A.5 CPS Proof 55
A.6 Defunctionalized Version 57
A.7 Defunctionalized Proof 61

B Appendix 2 Hornify **65**
B.1 Full Implementation 65
B.2 Evaluation Functions 68
B.3 Hornify Proof 71

Chapter 1

Introduction

This chapter presents a brief introduction to the work developed in this dissertation. Starting with the contextualization (Section 1.1), FACTOR project (Section 1.2), identification of the problem (Section 1.3), objective (Section 1.4) and 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 topic, not only for computer scientists but also for the general public. News about the discovery of new bugs and vulnerabilities are published almost every day, which is making the societal tolerance to software bugs 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 [32], 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 [46].

The ambition to develop completely bug-free programs has existed nearly as long as the field of computer science. Verifying every single program execution scenario is a challenge, it is impossible to imagine every single scenario and every single behavior of our programs. Although of this impossibility, programmers still make assumptions of 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 the program, under some defined specification, is correct. So to ensure that a program is free of bugs we should resort to it. However, developers have only habits for testing and almost none for program verification. For many, there are no methodologies or tools for verification on 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 [24]. FACTOR is funded by the Tezos Foundation [44] and aims to promote the use of OCaml 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.

1.3 Problem

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 and executable implementations, provably correct to stress the importance of sound practices.

These courses usually are either too informal or too formal when describing algorithms, neglecting a little that this is a course of Computer Science. If too informal, only pseudo-codes are provided, which hampers the understanding of all the aspects of the algorithm. 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.

In either case, there is an emphasis on paper-and-pencil proofs rather than on computational approaches. It is seldom the case where courses provide running algorithms (executable programs). Implementations of algorithms and tools that help students go through the resolutions of exercises are crucial, as they are an important pedagogical object. This is will be later discussed in Section 2.1.

Another aspect as mention is the lack of material about correction. As mention in section 2, program verification is an important aspect of software development, since it allows the development of bug-free software.

Most courses make only a small or no effort on showing the presented algorithms correction. The correction of these algorithms can also provide a deeper knowledge of the same ones. For instance, the Integrated Master in Computer Science 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, students can opt to do it or not.

Students of the Faculty of Science and Technology of NOVA University of Lisbon can finish a Computer Science Master Degree without knowledge about program verification, an important aspect that should be natural for Computer Scientists.

1.4 Objective

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

The objective is to give **supplementary** pedagogical material to a unit 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, the implementations should allow step-by-step executions and undoes, which adds the ability to slowly follow each step of the algorithm, traducing in a better algorithm apprenticeship.

Functional languages are specially suitable to handle symbolic computation and are based on mathematical functions. These type of language have a closer relationship to the mathematical definitions, making it a suitable language for these implementations.

To achieve step-by-step executions CPS (Continuation-passing style) will be used. Continuation-passing style is a particular way of defining and calling functions. The idea with continuation-passing style is instead of extract a value back, you send in your function to continue (or terminate) a program action. Functions written in CPS takes an extra argument called a callback, which is a function itself, so instead of returning the result of their computation, they call the callback on it [43].

Once the algorithm is implemented we need to verify its implementation. The verification 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 expected contributions are:

1. Clear identification of properties and specifications of propositional logic algorithms.
2. Have step-by-step executable implementation of the algorithms in a functional language.
3. Present the correspondent correctness criteria.
4. Verify the implementations using Why3.

The implementations, correctness criteria, and soundness proof will serve as pedagogical material to Computational Logic students.

1.6 Document Structure

This dissertation is organised as follows:

- Chapter 1 - Introduction contextualizes, identifies the problem, defines the objective and main contributions of the dissertation.
- Chapter 2 - State of the Art reports the aspects of Computational Logic unit, the existing tools to support this unit and the main proof-assistant tools.
- Chapter 3 - Background and Preliminaries does in-depth view about program verification, continuation-passing style and defunctionalization; also presents a simple proof in Why3.
- Chapter 4 - Conjunctive Normal Form Transformation Algorithm presents the implementation and verification of the algorithm that converts propositional formulae to CNF.
- Chapter 5 - Hornify presents the implementation and verification of the algorithm that converts CNF formulae to Horn Clauses.
- Chapter 6 - 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 an introduction to the context and problem of this dissertation was made, in which the importance of Computational Logic unit, the lack of tools and implementations of algorithms, and the little emphasis about program verification was mentioned.

This chapter presents the global overview of Computational Logic unit in Portugal in contrast of the ACM and IEEE recommendations and the actual tools to support this unit. Still in this chapter are presented and compared 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 [19] 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 [40]. 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 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 [40] and in the information available online.

FCT-NOVA

The Integrated Master in Computer Science has the course Computational Logic, that is a compulsory curricular unit, being taught in the 1st semester of the 2nd 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 [33]. The bibliographic's main reference is Language Proof and Logic (2nd edition), Dave Barker-Plummer, Jon Barwise and John Etchemendy, CSLI Publications, 2011

University of Beira Interior

The degree in Computer Science and Engineering offers a Computational Logic course, covering all the aspects of the syllabus. The references are Michael Huth and Mark Ryan. Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press, 2004; and J.B. Almeida, M.J. Frade, J.S. Pinto, and S. Melo de Sousa. Rigorous Software Development, An Introduction to Program Verification. Volume 103 of UTiCS. Springer-Verlag, first edition, 307 p. 52 illus. edition, 2011. 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) [34]

IST

The degree in Informatics Engineering and Computers has the course Logic for Programming that teaches propositional and first-order logic, resolution, and Prolog. The bibliographic references are the following: Lógica e Raciocínio., João P. Martins, College Publications, 2014; Mathematical Logic for Computer Science, M. Ben-Ari, Springer-Verlag, 2001; Logic in Computer Science: Modelling and Reasoning about Systems, M. Huth e M. Ryan, Cambridge University Press, 2004 [25].

University of Algarve

The degree in Informatics has a course on Logic and Computation that addresses Propositional and first-order logic (syntax, semantics, and deductive systems) in 50% of the syllabus. The bibliographic references are again Huth and Ryan and the classic Logic for Mathematicians, A. G. Hamilton, Cambridge University Press, 1988 [50].

University of Aveiro

The degree in Informatics Engineering does not have a specific course on logic. Propositional and First-Order Logics are part of the Discrete Mathematics course subject, composing about 1/6 of the syllabus. The bibliography's main reference is Tópicos de Matemática Discreta, J. S. Pinto, Universidade de Aveiro, 1999.

University of Minho

The degree in Informatics Engineering does not have a specific course; Propositional Logic is inserted in the course of Discrete Structures and composes about 1/6 of the syllabus. The bibliography references include Barwise and Etchemendy, the main one being Estruturas Discretas: textos de apoio, Jorge Picado, DMUC, 2008.

University of Lisbon

The degree in Informatics Engineering has a course on First-Order Logic. The main bibliography reference is again Barwise and Etchemendy

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. It covers Propositional and First-Order Logic - syntax, semantics, algorithms for satisfiability checking, deductive systems, and resolution. The course requires approval on the Discrete Mathematics course. The main bibliography reference is again Huth and Ryan, and the Lecture Notes of Nelma Moreira.

In conclusion, the Logic courses in the Computer Science degrees in Portugal mostly follow the ACM and IEEE recommendations, however in some cases with less emphasis and in the remaining coinciding in the content: Syntax, semantics, and deductive systems of Propositional and of first-order logic. The courses including a computational view (circa 50%) present satisfiability algorithms and the resolution method. Barwise & Etchemendy and Huth & Ryan are the most popular references [40].

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. 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 [7]. The program is more like a game where students can create worlds and make sentences about the constructed world, the program then will evaluate the sentences and give feedback to the user.

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

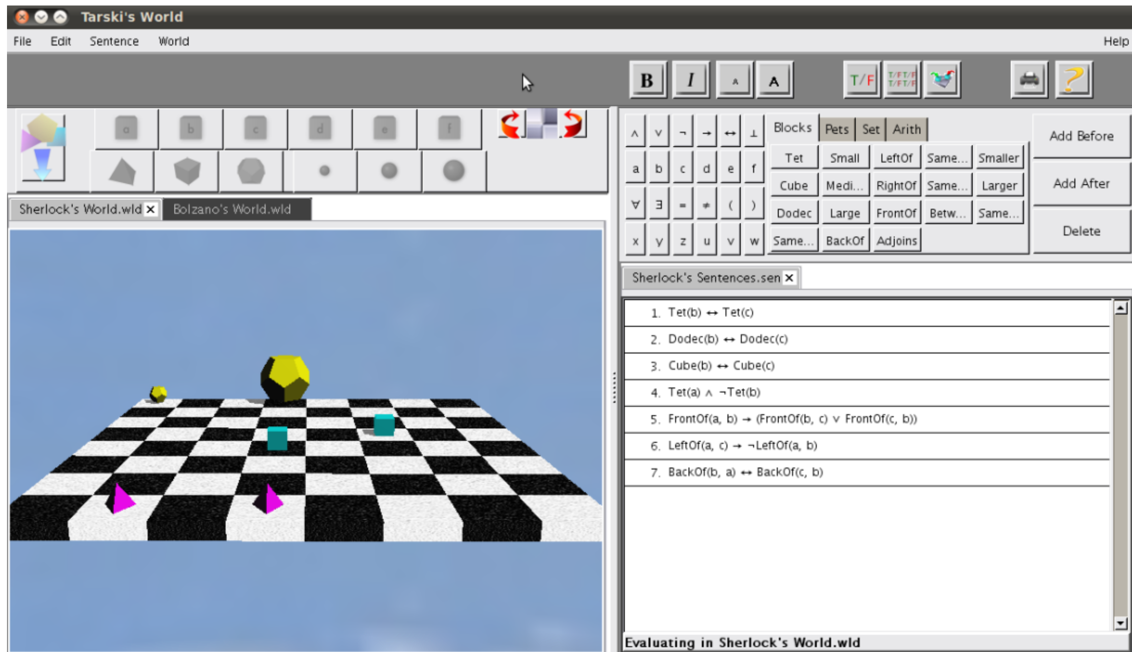

Figure 2.1: OpenProof (n.d.). Tarski's World main window. [image] Available at: https://ggweb.gradegrinder.net/support/manual/tarski.

2.2.2 Fitch

Fitch notation is a notational system for constructing formal proofs used in propositional logic and 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: *(taken 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: 1, 2]
5  |    |    not not P         [negation introduction: 2]
6  |
7  |    |__ not not P          [assumption, want P]
8  |    |    P                 [negation elimination: 5]
9  |
10 |    P iff not not P        [biconditional introduction: 1 - 4, 5 - 6]
```

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

Figure 2.2: Fitch-checker $A- > B, A- > C \therefore A- > (B \wedge C)$ proof. [image] Available at: https://github.com/OpenLogicProject/fitch-checker.

2.2.3 Boole

Boole [14] is a simple tool that helps students to build and verify truth tables.

Figure 2.3: Boole program

2.3 Proof-Assistant Tools

Proof assistant tools are computer systems to assist with the development of formal proofs by human-machine collaboration [26]. In this section, a brief description of some tools will be presented.

2.3.1 Why3 / Dafny

Why3 is a platform for deductive program verification [13]. The Why3 objective is to provide as much automation as possible, which distinguishes 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 [13]. This is an added value when the user wants to use different provers, rather than stick with 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.

Dafny is a programming language with built-in specification constructs, designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive data types [20]. Dafny is the closest tool to Why3 but has the downside that only supports the Z3.

2.3.2 Coq / Isabelle / Agda

Coq [8] 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 [45]. Similar to Why3, Coq can extract certified OCaml programs.

Isabelle/HOL [37] 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 [18]. Similar to the other theorem provers described, Isabelle/HOL also has his special ML programming language.

Agda [15] is a proof assistant and a dependently typed functional programming language. It is an interactive system for writing and checking proofs. It is based on intuitionistic type theory and has many similarities with other proof assistants based on dependent types, such as Coq [3]. However, it has no support for tactics [1].

2.3.3 CFML

CFML stands for Characteristic Formulae for ML. It is based on Characteristic Formulae. Characteristic formulae can be used to prove any true property of a program and can be

[1] A tactic replaces the goal with the subgoals it generates.

applied to current existing programs since it is not specific to any particular programming language. The characteristic formula of a program is a higher-order logic formula that gives a complete description of the semantics of the program without referring to its source code [16].

Shortly, CFML consists of a generator that parses OCaml code and produces characteristic formulae expressed as Coq axioms and a Coq library that provides tactics for manipulating characteristic formulae interactively [17].

2.3.4 Twelf

Twelf [39] is a language used to specify, implement, and prove properties of deductive systems such as programming languages and logics [48]. The Twelf theorem proving component is at an experimental stage and currently under active development [49].

2.3.5 Conclusions

This dissertation aims to increase the use of OCaml, proving that they are suitable for algorithm presentations and simpler proofs. The major of the tools offer some ML-like language. However, only Coq and Why3 can extract certified OCaml programs. The difference between Coq and Why3 is that Why3 is based on a first-order logic with inductive predicates and automatic provers, and Coq on an expressive theory of higher-order logic [18]. The work of this dissertation will reside on first-order algorithms with the focus that the proof must have as much automation as possible, making it easier for undergraduate students to understand. All these aspects turn Why3 on the top choice to this work.

CHAPTER 3

Background and Preliminaries

In Chapter 2 were presented some proof-assistant tools, where we concluded that Why3 was the ideal tool for our work. 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, we can 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 [28], so named because it is made up of three components: precondition, program and postcondition.

$$\{precondition\}\ P\ \{postcondition\}$$

The triple means that, if the precondition is true before initiation of the program P, then the resulting state will satisfy the postconditions.

Partial vs Total Correctness. The Hoare triple mention before is a partial correctness triple for specifying and reasoning about the behaviour of a program. Partial correctness triple because assuming that the precondition is true just before the function executes, then if the function terminates, the postcondition is true. In order to totally correct a program, 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}. However, although correct, this Hoare triple is not a precise as we might think. Especially, 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. Properly, if {precondition} P {postcondition} and for all the postcondition such that {precondition} P {postcondition}, the postcondition $\implies$ postcondition, then we are facing the strongest postcondition of S with respect to the precondition. The same way in the opposite direction. If {precondition) P {postcondition} and for all precondition such that {precondition} P {postcondition}, precondition $\implies$ precondition, then we are facing the weakest precondition `wp(P,postcondition)` of P with respect to the postcondition [5].

Edsger Dijkstra had a major role in the automation of deductive program verification with his work for weakest precondition calculus [22]. 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:

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

Properly, the validity of a Hoare triple {precondition} P {postcondition} is provable in Hoare logic for total correctness if and only if the first-order predicate below holds:

$$precondition \implies wp(P, postcondition)$$

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 of this dissertation, is one of these tools. However, there are more, for example, Dafny [31], VeriFast[30] and Viper[36].

The VCG takes as input a program along with the desired specification and generates a set of logical statements called verification conditions. Now comes to the second part of the verification process where we need to prove the validity of these logical statements. For that, we use theorem provers, a standard approach that is used to discharge verification conditions. If we want the most automation possible we should turn ourselves to automatic theorem provers. In specific, the family of SMT (Satisfability Modulo Theory) solvers [9], for example, Alt-Ergo [12], CVC4 [10] and Z3 [21].

3.2 Why3

A brief description of Why3 was presented in Section 2.3.1. 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 already ensured 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 }
  = ...
```

Here is the graphical interface of our proof:

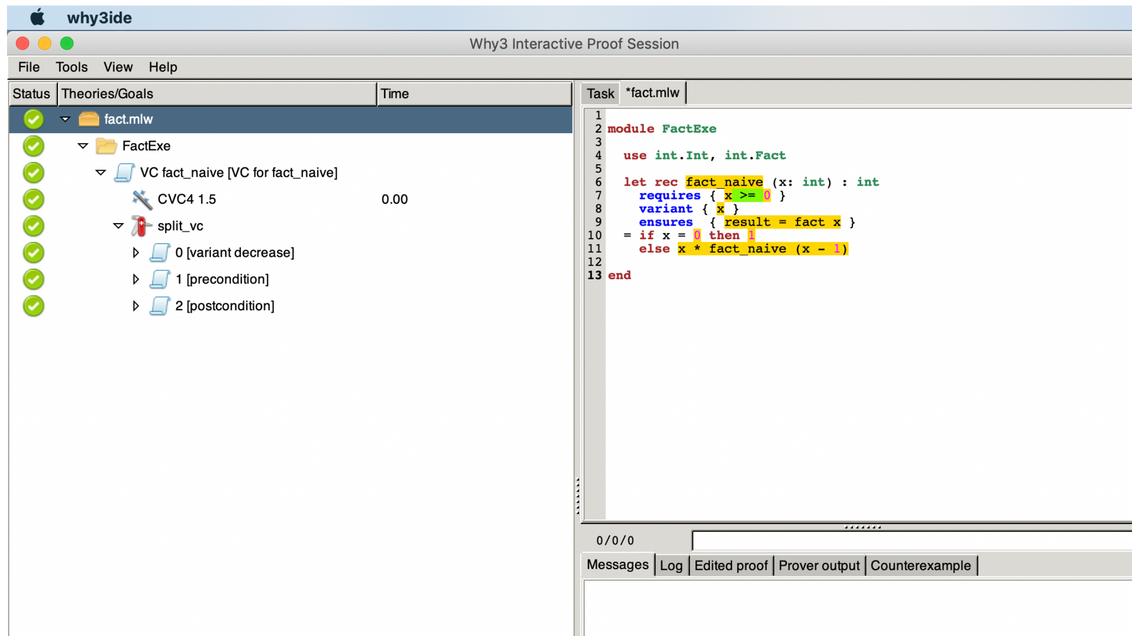

Figure 3.1: Proof of factorial function using Why3

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 [6]. 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.

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:

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

Let us compute the Factorial of 5:

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 (Final Iteration)

Practically, this can be seen as: (5 * (4 * (3 * (2 * (1 * 1)))))

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 [47].

The main advantage is the full control over control flow. In CPS there are no statements one after the other. 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 control flow will be useful. The continuation function can be used as a block, thus allowing to stop and return the execution.

3.4 Defuncionalization

Defunctionalization is a program transformation technique to convert high-order programs into first-order. Originally, introduced by Reynolds as a technique to transform a higher-order interpreter into a first-order one [41]. Recent studies uses this technique to derive abstract machines for different strategies of evaluation lambda-calculus from compositional interpreters [4, 38].

Let us consider the example of our factorial function to better understand the defuncionalization technique:

```
1 let rec fact_cps (x: int) (k: int → int) : int
2   = if x = 0 then k 1
3     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 defuncionalize this function, we need to represent in first-order this two anonymous functions. For that, we create a new algebraic type that captures the values of the free variables:

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

The Kid represents the identity function, thus does not have any free variable. The KFact represent 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 correspondent constructor:

```
1 let rec fact_cps (x: int) (k: int → int) : int
2   = if x = 0 then ...
3     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:

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

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

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

Transformation process. As seen above, the defuncionalization 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

Conjunctive Normal Form Transformation Algorithm

The Conjunctive Normal Form (CNF) [1] is common 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 [23, 27, 35], or informally, intended for Computer Science but with textual definitions in non-executable pseudo-code [11, 29]. The implementation of algorithms of this nature is a fundamental piece for learning and understanding them. Herein we present two implementations, formally verified in Why3, from a presentation as a recursive function of conversion algorithm to CNF: the first in direct style and the second with an explicit stack structure.

4.1 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:

$$
\begin{aligned}
G_p \triangleq \ \phi &::= \text{ł} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \\
\text{ł} &::= p \mid false
\end{aligned}
$$

The function T produces formulae in CNF, a formula in CNF is an element of the set J_p, defined as follows:

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

[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.

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

$$T(\phi) = \text{CNFC (NNFC (Impl_Free } (\phi)))$$

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)[2]
- 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:

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

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

4.2 Implementation

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

Analyzing the sets there is possible to observe that the ł grammar is present in all of them. Therefore, we created a general type `literal` representing this grammar:

```
1 type literal =
2   | FVar ident
3   | FConst bool
```

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

```
1 type formula =
2   | L literal
3   | FAnd   formula formula
4   | FOr    formula formula
5   | FImpl  formula formula
6   | FNeg   formula
```

The set H_p has literals, conjunctions, disjunctions, the primitive negation connective, don't has implications and is represented by the type `formula_wi`:

[2] A formula is in NNF if the negation operator is only applied to sub-formulae that are literals.

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

The set I_p has literals, conjunctions, disjunctions, the negation connective (in this case the negation connective can only be applied to literals) and is represented by the type `formula_nnf`:

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

The set J_p has literals, negation of literals, disjunctions, 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 =
  | FAnd_cnf formula_cnf formula_cnf
  | D disj

type disj =
  | FOr_cnf disj disj
  | FNeg_cnf literal
  | L_cnf literal
```

Functions. The function `Impl_Free` removes all the implications. It is recursively defined in the cases of the type `formula` and homomorphic, 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
        | FNeg phi1 → FNeg_wi (impl_free phi1)
        | FOr phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
        | FAnd phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
        | FImpl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
        | L  phi → L_wi phi
    end
```

The functions NNFC converts formulas 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
        | 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
```

The CNFC function converts formulas 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
        | FOr_nnf phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
        | FAnd_nnf phi1 phi2 → FAnd_cnf (cnfc phi1) (cnfc phi2)
        | FNeg_nnf literal → D (FNeg_cnf literal)
        | L_nnf literal → D (L_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 is the following:

```
let rec distr (phi1 phi2: formula_cnf) : 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)
        | D phi1, D phi2 → D (FOr_cnf phi1 phi2)
    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 full implementation code is in Appendix A.1.

4.3 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 valuation 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:

```
type valuation = ident → bool

function eval_literal (v: valuation) (f: literal) : bool
  = match f with
      | FVar x → v x
      | FConst b → b
    end

function eval (v: valuation) (f: formula) : bool
  = match f with
        | L phi1      → eval_literal v phi1
        | FAnd f1 f2  → eval v f1 && eval v f2
        | FOr f1 f2   → eval v f1 || eval v f2
        | FImpl f1 f2 → (eval v f1 → eval v f2)
        | FNeg f      → not (eval v f)
    end
```

This function takes an argument of type `valuation` assigning a value of type `bool`[3] to each variable of the formula, receives the formula to evaluate and returns a value of type `bool`. For the base constructor is `L` a literal, the Boolean value of the variable and 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 WhyML Boolean operation. The valuation function for the type of formulae `formula_wi` is similar.

The rest of the evaluation functions is in Appendix A.2.

[3] bool is the Boolean type of WhyML.

4.4 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 valuation 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 constructors, 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 an auxiliary lemma:

```
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 }
  = ...
```

However, 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 complete code of the specification is in Appendix A.3.

4.5 Continuation-Passing Style

As mention in Section 3.3, *Continuation-Passing Style* (CPS) is a programming style where the control is passed explicitly in the form of a continuation.

Process transformation into CPS. The transformation is performed mechanically according to the following steps:

- Given a function of type `t' → t`, we add an argument which represents the continuation (a function of type `t → 'a`) and change the return type of the function to `'a`.
- For the base cases instead of returning the desired values, we apply these values to the continuation function.
- For the remaining cases, we start by making a recursive call and construct the continuations with the rest of the computation.
- We add a `main` function that calls the function in CPS with the identity function as a continuation.

We apply now this process to the functions presented in the previous section. With respect to the `Impl_Free` function:

1. We add an argument to the function of type `formula_wi → 'a` and change the return type to `'a`:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
```

2. For the base cases, we apply the desired values to the continuation function.

```
    | L phi → k (L_wi phi)
```

3. For the remaining cases, we start with a recursive call and define the continuations with the rest of the computation:

```
    | FNeg phi1 → impl_free_cps phi1 (fun con → k (FNeg_wi con))
    | FOr phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (
    fun con1 → k (FOr_wi con con1)))
    | FAnd phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (
    fun con1 → k (FAnd_wi con con1)))
    | FImpl phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2
    (fun con1 → k (FOr_wi (FNeg_wi con) con1)))
```

4. Finally, we create the `main` function that calls the function in CPS with the identity function as a continuation:

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

We obtain the CPS version of the remaining functions in a similar way to this process (the complete code 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, since these are pure and total functions, as specification, *i.e.*, we simply assure that the result is equal to the result of the functions in direct style, applied to the continuation.

For the `Impl_Free` function in CPS, it is enough to ensure that the result is equal 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 a 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 to the specification of the Impl_Free function referred above (Appendix A.5).

4.6 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. Both implementations were 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 the H_p, which increases the complexity of the proof. We first started following the structure of the algorithm that is present in the available slides 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, which gave us some challenges and led us to the defunctionalization technique.

4.6.1 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 is not possible to ensure the Normal Negation Form and Conjunctive Normal Form. So to ensure the NNF and CNF, we create two well-formulated 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 there 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 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 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
```

Continuation-Passing Style. For the CPS functions, there are only differences in the CNFC function. It is necessary to prove its preconditions. In particular, it is necessary to prove that the input formula is in NNF.

A proof obligation is generated regarding the validity of the precondition 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 wf_negations_of_literals predicate into a new type (an invariant type):

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

Since the return type of the function is changed, the proof of the post-conditions now involves the comparison of two invariant types, which raises some interesting challenges.

Difficulties preventing the 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 behavior:

```
1 lemma types: forall x y. x.cnfc_formula = y.cnfc_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 `cnfc_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 (=).

```
1 predicate (==) (t1 t2: cnfc_type) = t1.cnfc_formula = t2.cnfc_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 decidability, operates on a first-order language, the solution is to "go back"to first order. The defunctionalization technique emerged as a possible approach.

4.6.2 Defunctionalization

Defunctionalization is a program transformation technique to convert high-order programs into first-order ones [42].

Transformation process. 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 a 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 represent the function continuations in first-order:

```
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
```

The constructor KImpl_id represents the identity function, the constructor KImpl_Neg represents the continuation of the case of the constructor FNeg_wi. As the remaining cases contain two continuation functions, 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 then replace the continuations with the new representation of the continuations:

```
let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
= match phi with
  | FNeg phi1 → impl_free_desf_cps phi1 (KImpl_Neg k phi1)
  | FOr phi1 phi2 → impl_free_desf_cps phi1 (KImpl_OrLeft phi2 k)
  | FAnd phi1 phi2 → impl_free_desf_cps phi1 (KImpl_AndLeft phi2 k)
  | FImpl phi1 phi2 → impl_free_desf_cps phi1 (KImpl_ImplLeft phi2 k)
  ...
end
```

The next step is to introduce an apply function, and replace the applications to the continuation:

```
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_desf_cps phi1 (KImpl_OrRight k phi)
  | KImpl_OrRight k phi2 → impl_apply (FOr_wi phi2 phi) k
  | KImpl_AndLeft p k → impl_free_desf_cps p (KImpl_AndRight k phi)
  | KImpl_AndRight k phi2 → impl_apply (FAnd_wi phi2 phi) k
  | KImpl_ImplLeft phi1 k → impl_free_desf_cps phi1 (KImpl_ImplRight k phi)
  | KImpl_ImplRight k phi2→ impl_apply (FOr_wi (FNeg_wi phi2) phi) k
end

```

```
13 let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
14 = match phi with
15   ...
16   | FConst phi → impl_apply (FConst_wi phi) k
17   | FVar phi → impl_apply (FVar_wi phi) k
18 end
```

The result of the application of the defunctionalization transformation to the remaining functions of the T algorithm in CPS is 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` [38].

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 a predicate performs case analysis on the continuation type and for each constructor, we copy the post-condition present in the corresponding abstraction [38]. For instance, for the `Impl_Free` function, we provide the following specification

```
1 let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
2   ensures { impl_post k (impl_free phi) result }
3  = ...
4
5 with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
6   ensures { impl_post k phi result }
7  = ...
```

The impl_post predicate is:

```
1 predicate impl_post (k: impl_kont) (phi result: formula_wi)
2 = match k with
3   | KImpl_Id → let x = phi in x = result
4   | KImpl_Neg k phi1 → let neg = phi in impl_post k (FNeg_wi phi) result
5   | KImpl_OrLeft phi1 k → let hl = phi in impl_post k (FOr_wi phi
6     (impl_free phi1)) result
7   | KImpl_OrRight k phi2 → let hr = phi in impl_post k (FOr_wi phi2 hr) result
8   | KImpl_AndLeft phi1 k → let hl = phi in impl_post k (FAnd_wi phi
9     (impl_free phi1)) result
10  | KImpl_AndRight k phi2 → let hr = phi in impl_post k
11    (FAnd_wi phi2 hr) result
12  | KImpl_ImplLeft phi1 k → let hl = phi in impl_post k (FOr_wi
13    (FNeg_wi phi) (impl_free phi1)) result
14  | KImpl_ImplRight k phi2→ let hr = phi in impl_post k (FOr_wi
```

Figure 4.1: Statistical result of defunctionalization proof obligations

```
15      (FNeg_wi phi2) hr) result
16  end
```

The proof of the post-conditions of the NNFC and CNFC defunctionalized functions is similar to the proof of the Impl_Free function (Appendix A.7). However, similar to the CPS proof, for the CNFC function, we have to prove its preconditions. For this we have created the invariant type wf_cnfc_kont with the well-formulated predicate wf_cnfc_kont as invariant:

```
1  type wf_cnfc_kont = {
2      cnfc_k: cnfc_kont;
3  } invariant { wf_cnfc_kont cnfc_k }
4    by { cnfc_k = KCnfc_Id }
```

Note that in this well-formulated predicate we just want to ensure the CNF for the formulae that are already converted. Given that the formulae are only converted in the right continuation, these and only these feature the wf_conjunctions_of_disjunctions predicate:

```
1  predicate wf_cnfc_kont (phi: cnfc_kont)
2  = match phi with
3    | KCnfc_Id → true
4    | KCnfc_OrLeft phi k → wf_negations_of_literals phi ∧ wf_cnfc_kont k
5    | KCnfc_OrRight k phi → wf_negations_of_literals phi ∧
       wf_conjunctions_of_disjunctions phi ∧ wf_cnfc_kont k
6    | KCnfc_AndLeft phi k → wf_negations_of_literals phi ∧ wf_cnfc_kont k
7    | KCnfc_AndRight k phi → wf_negations_of_literals phi ∧
       wf_conjunctions_of_disjunctions phi ∧ wf_cnfc_kont k
8  end
```

Lastly, the proof of the T function turns out to be similar to the direct-style proof referenced in Page 25:

```
1  let t (phi: formula) : formula_wi
2    ensures { forall v. eval v phi = eval_wi v result }
3    ensures { wf_negations_of_literals result }
4    ensures { wf_conjunctions_of_disjunctions result }
5  = cnfc_desf_main(nnfc_desf_main(impl_desf_main phi))
```

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 Figure 4.1.

Chapter 5

Hornify

The Horn algorithm is a simple and easy solution to determine with polynomial complexity if a given propositional formula is satisfactory or contradictory. However, the algorithm works only for a particular set of formulae - the Horn Clauses. There are few presentation of this transformation algorithm and usually imperative, we could not find any functional presentation neither its implementation. This section presents the transformation algorithm from CNF to Horn Clause, including the implementation and verification of the algorithm.

5.1 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.
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. $L \equiv \top \to L$
2. $\bigvee_{i=1}^{n} \neg L_i \equiv (\bigvee_{i=1}^{n} L_i) \to \bot$
3. $\bigvee_{i=1}^{n} \neg L_i \vee L \equiv (\bigvee_{i=1}^{n} L_i) \to L$

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

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

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

Where Li 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:

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

5.2 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 by the following grammar:

$$\begin{aligned}
&\phi ::= \phi \wedge \phi \mid \tau && \textit{(formula_cnf)}\\
&\tau ::= l \mid \neg l \mid \tau \vee \tau && \textit{(disjunction)}\\
&ł ::= p \mid false && \textit{(literal)}
\end{aligned}$$

The main function is the `hornify` and has the following signature:

$$\text{hornify} : formula_cnf \rightarrow hornformula$$

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

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

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

$$\text{hornify} : formula_cnf \rightarrow basichornformula$$

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

This functions follows the properties presented in the Section 5.1. The main case of the function is when the formula is a disjunction. In this case, the function calls the `hornify_aux` functions to get all the positive and negative literals into a set, then uses the `conjunction` function to build the conjunction of the negative literals and the `getPositive` 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:

$$\text{hornify_aux} : disjunction * set * set \rightarrow set * set$$

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

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

$$\text{conjunction} : set \rightarrow leftside$$

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

The `getPositive` functions is responsible to build the right side of the implication. Given a empty set or a set with only one positive literal, returns one positive literal ($\varnothing$ or itself):

$$\text{getPositive} : set \rightarrow rightside$$

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

5.3 Implementation

Firstly, it is necessary to define the specific formulae types according to the grammar defined in Section 5.1:

- $\psi ::= \mu \mid \psi \wedge \psi$:

```
1 type hornclause =
2   | HBasic basichornclause
3   | HAnd hornclause hornclause
```

- $\mu ::=\ \chi \rightarrow \omega$:

```
type basichornclause =
  | BImpl leftside rightside
```

- $\chi ::= \top \mid \alpha$:

```
type leftside =
  | LTop
  | LPos positive
```

- $\alpha ::= p \mid \bot \mid \alpha \wedge \alpha$:

```
type positive =
  | PLCBottom
  | PLCVar ident
  | PLCAnd positive positive
```

- $\omega ::= p \mid \bot \mid \top$

```
type rightside =
  | RBottom
  | RTop
  | RVar ident
```

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

```
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
```

As mention before, the principal case of the function is when the formula is a disjunction. In this case, the hornify_aux functions is called in order to get all the positive and negative literals and uses the conjunction and getPositive functions to build the implication:

```
let getBasicHorn (phi: clause_cnf) : basichornclause
  = match phi with
      | DLiteral (LVar x) → BImpl (LTop) (RVar x)
      | DLiteral (LBottom) → BImpl (LTop) (RBottom)
      | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RBottom)
      | DNeg_cnf (LBottom) → BImpl (LTop) (RTop)
      | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
                            BImpl (conjunction s) (getPositive p)
    end
```

The conjunction function uses an built-in 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:

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

The getPositive functions returns one positive literal – ⊥ if the option is None or itself if it is Some x:

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

The hornify_aux function goes through the disjunction combinations, positively adds the negative literals to the input set and the positive literal to the option type. This function can also raise the MoreThanOnePositive exception if there is more than one positive literal in the formula:

```
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 (convertLiteralToPLC nl1) (add (convertLiteralToPLC 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 (convertLiteralToPLC 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 processes the disjunction formula when one is positive and the other is negative. If the option 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 adds the negative literal to the set and sets the `option` with 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 (RBottom) in
                  match nl with
                    | LBottom → ((add (PLCBottom) s), rbottom)
                    | LVar x → ((add (PLCVar x) s), rbottom)
                  end
      | LVar x → let rvar = Some (RVar x) in
                  match nl with
                    | LBottom → ((add (PLCBottom) s), rvar)
                    | LVar x → ((add (PLCVar x) s), rvar)

                  end
   end
```

The complete code is in Appendix B.1

5.4 Verification

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

Less trivial cases. In some cases, especially when the formula has more than one input and output, the equivalence of the formulae 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 $\vee$ $(\vee_{i=1}^{n} \neg E_i)$ $\vee$ p = $(\vee_{i=1}^{n} \neg R_i)$ $\vee$ rp
3. phi $\vee$ $(\vee_{i=1}^{n} \neg E_i)$ $\vee$ p = $(\wedge i = 1^n R_i) \rightarrow$ 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, it is a must to ensure that the input formula (phi) is effectively 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 v. eval_domain v phi s p = eval_codomain v rs rp }
  ensures{ forall v. eval_domain v phi s p = implb (eval_conjunction_set v rs) (eval_optionrightside
     v rp) }
  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 side of the equality of property 2):

```
function eval_domain (v: valuation) (phi: clause_cnf) (s: set) (p: option rightside) : bool
  = eval_disj_cnf v phi || eval_set v s || eval_optionrightside v p

function eval_codomain (v: valuation) (s: set) (p: option rightside) : bool
  = eval_set v s || eval_optionrightside v p
```

In the processCombination and addLiterals functions it is needed to ensure, once again, that the output set is not empty and that the domain is equivalent of the codomain. This obligation can be traduced into the following property:

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

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:

```
ensures{ (not is_empty rs) }
ensures { forall v. (eval_literal v pl || not (eval_literal v nl) || eval_set v s ||
    eval_optionrightside v p)
          = (eval_set v rs || eval_optionrightside v rp) }
```

The full specification is in Appendix B.3.

Auxiliary Lemma and Axiom. Evaluation functions some times need to evaluate all the elements of a given set. Unfortunately, there are some properties that the type set of Why3 cannot assure. For example, the De Morgan's laws over elements of a set.

We defined the deMorgan lemma following the structural induction proof method. The objective with this lemma is to assure that the evaluation of a disjunction over all the elements of a set is equal to the negation of the evaluation of a conjunction over all the elements of a set, but this time with the elements negated:

```
let rec lemma deMorgan (v: valuation) (s: fset positive)
    requires{ not is_empty s }
    ensures{ eval_set v s = not eval_conjunction_set v s }
    variant{ FS.cardinal s }
  = if FS.cardinal s > 1 then
      let x = FS.pick s in
        deMorgan v (FS.remove x s)
```

Other property that the set type of Why3 cannot guarantee is that the evaluation of a certain set with an added element x is equal to the evaluation of the element x with the evaluation of the rest of the set. Since this property is crucial to the proof, we defined this theory. Starting with an empty set, if we add an element to the set, the evaluation is equal to evaluation of the element x itself.

```
lemma evalemptyset_aux:
    forall v s x. (is_empty s) → eval_set v (FS.add x s) = ((not eval_positive
    v x) || eval_set v s)
```

This lemma was naturally processed. However, when the set is not empty, it is not possible to prove. Several tries to prove this lemma were made but unfortunately with no progress. Since this lemma is not strong, we decided to put it as an axiom in order to finish the proof:

```
axiom evalset_aux:
    forall v s x. (not (is_empty s)) → eval_set v (FS.add x s) = ((not
    eval_positive v x) || eval_set v s)
```

5.5 Conclusions and Observations

There is few presentations of this algorithm and when presented it is usual imperative. We present herein a new formulation of the algorithm. This Horn clauses are then used in the Horn algorithm.

The algorithm 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 a bit more complex. We present them in a functional language, showing that given their properties the implementation and specification is less complex and related to the presentation of the algorithm.

Chapter 6

Conclusions

The objective of this dissertation is to contribute to the development of pedagogical material to support the Computational Logic unit, through the implementation and verification of standard and classical algorithms of Computational Logic. The focus of this work was that the implementation and verification should be adequate for undergraduate students.

In the Chapter 4 and 5 were presented the implementation and verification of the Conjunctive Normal Form transformation algorithm and the algorithm that transforms the Conjunctive Normal Form formulae into Horn Clauses, respectively. It is possible to observe that functional languages such as OCaml allow close implementations of mathematical definitions without sacrificing clarity and rigour. These make them adequate to be pedagogically used as an aid to the study and understanding of algorithms.

Also, the proofs of the implementations are concise. We must ensure that the evaluation of the domain of our functions is equal to the evaluation of the co-domain. It is simple to understand the concept, however, in some cases derived from Why3 limitations, it led to a little complex solution. In the Hornify algorithm, to finish the proof we forced an axiom instead of a lemma, the desired approach would be to stick with the lemma and somehow prove it. Due to time factors, this was not possible, but we are intended to keep trying. In the CNF algorithm, the previous work (work following to structure of the algorithm the way it is taught in the Computational Logic unit) led us to hit rock bottom and develop another implementation using defunctionalization. The defunctionalization was a pleasant surprise, which the transformation process and his proof were a natural process.

Bibliography

[1] W. Ackermann. "Frederic Brenton Fitch. Symbolic logic. An introduction." In: *The journal of symbolic logic* (1952).

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

[3] *Agda*. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[4] 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*. PPDP '03. Uppsala, Sweden: ACM, 2003, pp. 8–19. ISBN: 1-58113-705-2. DOI: 10.1145/888251.888254. URL: http://doi.acm.org/10.1145/888251.888254.

[5] J. Aldrich. "Lecture Notes: Hoare Logic." In: *Analysis of Software Artifacts* ().

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

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

[8] 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." In: (1997).

[9] C. Barrett and C. Tinelli. "Satisfiability modulo theories." In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[10] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. "Cvc4." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177.

[11] M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012. ISBN: 978-1-4471-4128-0. URL: https://doi.org/10.1007/978-1-4471-4129-7.

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

[13] 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.

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

[15] A. Bove, P. Dybjer, and U. Norell. "A brief overview of Agda–a functional language with dependent types." In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 73–78.

[16] A. Charguéraud. "Characteristic formulae for the verification of imperative programs." In: *ACM SIGPLAN Notices*. ACM. 2011.

[17] A. Charguraud. URL: http://www.chargueraud.org/softs/cfml/.

[18] R. Chen, C. Cohen, J.-J. Levy, S. Merz, and L. Thery. "Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle." In: *Unpublished* (2018).

[19] A. CS2013. *Computer Science Curricula 2013. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. 2013. URL: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

[20] *Dafny: A Language and Program Verifier for Functional Correctness*. URL: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

[21] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[22] E. W. Dijkstra. "Guarded commands, nondeterminacy, and formal derivation of programs." In: *Programming Methodology*. Springer, 1978, pp. 166–175.

[23] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972. ISBN: 978-0-12-238450-9.

[24] *FACTOR: Functional ApproaCh Teaching pOrtuguese couRses*. URL: http://ctp.di.fct.unl.pt/FACTOR/.

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

[26] H. Geuvers. "Proof assistants: History, ideas and future." In: *Sadhana* (2009).

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

[28] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* (1969).

[29] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)* Cambridge University Press, 2004.

[30] 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*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–55. ISBN: 978-3-642-20398-5.

[31] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[32] 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*. 1996. URL: http://zoo.cs.yale.edu/classes/cs422/2010/bib/lions96ariane5.pdf.

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

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

[35] E. Mendelson. *Introduction to mathematical logic (3. ed.)* Chapman and Hall, 1987. ISBN: 978-0-412-06971-0.

[36] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by B. Jobstmann and K. R. M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5.

[37] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[38] M. Pereira. "Desfuncionalizar para Provar." In: *CoRR* abs/1905.08368 (2019). arXiv: 1905.08368. URL: http://arxiv.org/abs/1905.08368.

[39] F. Pfenning and C. Schürmann. "System description: Twelf—a meta-logical framework for deductive systems." In: *International Conference on Automated Deduction*. Springer. 1999, pp. 202–206.

[40] A. Ravara. "Objectives, syllabus, contents and teaching and assessment methods." In: *Report on the Curricular Unit Computational Logic* (2018).

[41] J. C. Reynolds. "Definitional interpreters for higher-order programming languages." In: *College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects. 13.* (1998). URL: https://surface.syr.edu/lcsmith_other/13.

[42] J. C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages." In: vol. 11. 4. 1998, pp. 363–397. URL: https://doi.org/10.1023/A:1010027404223.

[43] G. J. Sussman and G. L. Steele. "Scheme: A interpreter for extended lambda calculus." In: *Higher-Order and Symbolic Computation* (1998).

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

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

[46] M Thomas and H Thimbleby. "Computer Bugs in Hospitals: An Unnoticed Killer." In: (2018). URL: http://www.harold.thimbleby.net/killer.pdf.

[47] H. Trefftz. "Continuation–passing style in ML." In: *Computer Science Department Maharishi University of Management* (2002).

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

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

[50] *University of Algarve*. 2019. URL: https://www.ualg.pt/en/curso/1478.

Appendix 1 CNF Transformation Algorithm

A.1 Full Implementation

```
module Formula

  type ident

  type literal =
          | FVar ident
          | FConst bool

  type formula =
          | L literal
          | FAnd   formula formula
          | FOr    formula formula
          | FImpl  formula formula
          | FNeg   formula

  type formula_wi =
          | L_wi literal
          | FAnd_wi formula_wi formula_wi
          | FOr_wi formula_wi formula_wi
          | FNeg_wi formula_wi

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

```
27    type disj =
28          | FOr_cnf disj disj
29          | FNeg_cnf literal
30          | L_cnf literal
31
32    type formula_cnf =
33          | FAnd_cnf formula_cnf formula_cnf
34          | D disj
35
36  end
37
38  module T
39
40    use Formula, Valuation, Size, int.Int
41
42    let rec function impl_free (phi: formula) : formula_wi
43    = match phi with
44          | FNeg phi1 → FNeg_wi (impl_free phi1)
45          | FOr phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
46          | FAnd phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
47          | FImpl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
48          | L  phi → L_wi phi
49      end
50
51    let rec function nnfc (phi: formula_wi)
52    = match phi with
53          | FNeg_wi (FNeg_wi phi1) → nnfc phi1
54          | FNeg_wi (FAnd_wi phi1 phi2) → FOr_nnf (nnfc (FNeg_wi phi1)) (nnfc (
      FNeg_wi phi2))
55          | FNeg_wi (FOr_wi phi1 phi2) → FAnd_nnf (nnfc (FNeg_wi phi1)) (nnfc (
      FNeg_wi phi2))
56          | FNeg_wi (L_wi phi1) → FNeg_nnf (phi1)
57          | FOr_wi phi1 phi2 → FOr_nnf (nnfc phi1) (nnfc phi2)
58          | FAnd_wi phi1 phi2 → FAnd_nnf (nnfc phi1) (nnfc phi2)
59          | L_wi phi1 → L_nnf phi1
60    end
61
62    let rec function distr (phi1 phi2: formula_cnf)
63    = match phi1, phi2 with
64          | FAnd_cnf phi11 phi12, phi2 → FAnd_cnf (distr phi11 phi2) (distr phi12
       phi2)
65          | phi1, FAnd_cnf phi21 phi22 → FAnd_cnf (distr phi1 phi21) (distr phi1
      phi22)
66          | D phi1, D phi2 → D (FOr_cnf phi1 phi2)
67      end
```

```
68
69   let rec function cnfc (phi: formula_nnf)
70   = match phi with
71         | FOr_nnf phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
72         | FAnd_nnf phi1 phi2 → FAnd_cnf (cnfc phi1) (cnfc phi2)
73         | FNeg_nnf literal → D (FNeg_cnf literal)
74         | L_nnf literal → D (L_cnf literal)
75   end
76
77   let t (phi: formula) : formula_cnf
78   = cnfc (nnfc (impl_free phi))
79
80 end
```

A.2 Evaluation Functions

```
1   let function eval_literal (v: valuation) (f: literal) : bool
2   = match f with
3       | FVar x → v x
4       | FConst b → b
5     end
6
7   let rec function eval (v: valuation) (f: formula) : bool
8     variant { f }
9   = match f with
10      | L phi1      → eval_literal v phi1
11      | FAnd f1 f2  → eval v f1 && eval v f2
12      | FOr f1 f2   → eval v f1 || eval v f2
13      | FImpl f1 f2 → not (eval v f1) || eval v f2
14      | FNeg f      → not (eval v f)
15    end
16
17  let rec function eval_wi (v: valuation) (f: formula_wi) : bool
18    variant { f }
19  = match f with
20      | L_wi phi1 → eval_literal v phi1
21      | FAnd_wi f1 f2  → eval_wi v f1 && eval_wi v f2
22      | FOr_wi f1 f2   → eval_wi v f1 || eval_wi v f2
23      | FNeg_wi f      → not (eval_wi v f)
24    end
25
26  let rec function eval_nnf (v: valuation) (f: formula_nnf) : bool
27    variant{ f }
28  = match f with
29      | FAnd_nnf f1 f2 → eval_nnf v f1 && eval_nnf v f2
```

```
30         | FOr_nnf f1 f2 → eval_nnf v f1 || eval_nnf v f2
31         | FNeg_nnf literal → not (eval_literal v literal)
32         | L_nnf literal → eval_literal v literal
33       end
34
35     let rec function eval_disj (v: valuation) (f: disj) : bool
36       variant{ f }
37     = match f with
38         | FOr_cnf f1 f2 → eval_disj v f1 || eval_disj v f2
39         | FNeg_cnf literal → not (eval_literal v literal)
40         | L_cnf literal → eval_literal v literal
41       end
42
43     let rec function eval_cnf (v: valuation) (f: formula_cnf) : bool
44       variant{ f }
45     = match f with
46         | FAnd_cnf f1 f2 → eval_cnf v f1 && eval_cnf v f2
47         | D disj → eval_disj v disj
48       end
```

A.3 Direct Style Proof

```
1   let rec function impl_free (phi: formula) : formula_wi
2     variant{ phi }
3     ensures{ forall v. eval v phi = eval_wi v result }
4   = ...
5
6   let rec function nnfc (phi: formula_wi)
7     variant{ size phi }
8     ensures{ (forall v. eval_wi v phi = eval_nnf v result)}
9   = ...
10
11  let rec function distr (phi1 phi2: formula_cnf)
12    variant{ size_cnf phi1 + size_cnf phi2 }
13    ensures{ (forall v. ((eval_cnf v phi1 || eval_cnf v phi2) =
14          eval_cnf v result))}
15  = ...
16
17  let rec function cnfc (phi: formula_nnf)
18    variant{ phi }
19    ensures{ (forall v. eval_nnf v phi = eval_cnf v result) }
20  = ...
```

```
21 let t (phi: formula) : formula_cnf
22   ensures{ (forall v. eval v phi = eval_cnf v result)}
23 = ...
```

A.4 CPS Version

```
module T_CPS
  use Formula, Valuation, T, Size, int.Int

  let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  = match phi with
        | FNeg phi1 → impl_free_cps phi1 (fun con → k (FNeg_wi con))
        | FOr phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (
    fun con1 → k (FOr_wi con con1)))
        | FAnd phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (
    fun con1 → k (FAnd_wi con con1)))
        | FImpl phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2 (
    fun con1 → k (FOr_wi (FNeg_wi con) con1)))
        | L phi → k (L_wi phi)
    end

  let impl_free_main (phi: formula) : formula_wi
  = 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
  = 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)))
        | D phi1, D phi2 → k (D (FOr_cnf phi1 phi2))
    end
```

```
36   let distr_main (phi1 phi2: formula_cnf) : formula_cnf
37   = distr_cps phi1 phi2 (fun x → x)
38
39
40   let rec cnfc_cps (phi: formula_nnf) (k: formula_cnf → 'a) : 'a
41   = match phi with
42         | FOr_nnf phi1 phi2 → cnfc_cps phi1 (fun con → cnfc_cps phi2 (fun con1
       → distr_cps con con1 k))
43         | FAnd_nnf phi1 phi2 → cnfc_cps phi1 (fun con → cnfc_cps phi2 (fun
     con1 → k (FAnd_cnf con con1)))
44         | FNeg_nnf literal → k (D (FNeg_cnf literal))
45         | L_nnf literal → k (D (L_cnf literal))
46   end
47
48   let cnfc_main (phi: formula_nnf) : formula_cnf
49   = cnfc_cps phi (fun x → x)
50
51   let t_main (phi: formula) : formula_cnf
52   = cnfc_cps (nnfc_cps (impl_free_cps (phi) (fun x → x)) (fun x → x)) (fun x
     → x)
53
54 end
```

A.5 CPS Proof

```
1  module T_CPS
2
3    use Formula, Valuation, T, Size, int.Int
4
5    let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
6      variant{ phi }
7      ensures{ result = k (impl_free phi) }
8    = ...
9
10   let impl_free_main (phi: formula) : formula_wi
11     ensures{forall v. eval v phi = eval_wi v result}
12   = ...
13
14   let rec nnfc_cps (phi: formula_wi) (k: formula_nnf → 'a) : 'a
15     variant{ size phi }
16     ensures{ result = k (nnfc phi) }
17   = ...
```

```
18    let nnfc_main (phi: formula_wi) : formula_nnf
19      ensures{(forall v. eval_wi v phi = eval_nnf v result)}
20    = ...
21
22    let rec distr_cps (phi1 phi2: formula_cnf) (k: formula_cnf → 'a) : 'a
23      variant{ size_cnf phi1 + size_cnf phi2 }
24      ensures{ result = k (distr phi1 phi2) }
25    = ...
26
27    let distr_main (phi1 phi2: formula_cnf) : formula_cnf
28      ensures { (forall v. ((eval_cnf v phi1 || eval_cnf v phi2) = eval_cnf v
      result)) }
29    = ...
30
31    let rec cnfc_cps (phi: formula_nnf) (k: formula_cnf → 'a) : 'a
32      variant{ phi }
33      ensures{ result = k (cnfc phi)}
34    = ...
35
36    let cnfc_main (phi: formula_nnf) : formula_cnf
37      ensures{ (forall v. eval_nnf v phi = eval_cnf v result) }
38    = ...
39
40    let t_main (phi: formula) : formula_cnf
41      ensures{ (forall v. eval v phi = eval_cnf v result) }
42    = ...
43
44  end
```

A.6 Defunctionalized Version

```
module Desfunctionalization

  use Formula, Valuation, Predicates, T, int.Int

  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_wi
    | Knnfc_negorleft formula_wi nnfc_kont
    | Knnfc_negorright nnfc_kont formula_wi
    | Knnfc_andleft formula_wi nnfc_kont
    | Knnfc_andright nnfc_kont formula_wi
    | Knnfc_orleft formula_wi nnfc_kont
    | Knnfc_orright nnfc_kont formula_wi

  type distr_kont =
    | KDistr_Id
    | KDistr_Left formula_wi formula_wi distr_kont
    | KDistr_Right distr_kont formula_wi

  predicate wf_distr_kont (phi: distr_kont) = match phi with
    | KDistr_Id → true
    | KDistr_Left phi1 phi2 k →
        wf_negations_of_literals phi1 ∧ wf_conjunctions_of_disjunctions phi1 ∧
        wf_negations_of_literals phi2 ∧ wf_conjunctions_of_disjunctions phi2 ∧
        wf_distr_kont k
    | KDistr_Right k phi →
        wf_negations_of_literals phi ∧ wf_conjunctions_of_disjunctions phi ∧
        wf_distr_kont k
    end
```

```
42  type wf_distr_kont = {
43    distr_k: distr_kont;
44  } invariant { wf_distr_kont distr_k }
45    by { distr_k = KDistr_Id }
46
47  type cnfc_kont =
48    | KCnfc_Id
49    | KCnfc_OrLeft formula_wi cnfc_kont
50    | KCnfc_OrRight cnfc_kont formula_wi
51    | KCnfc_AndLeft formula_wi cnfc_kont
52    | KCnfc_AndRight cnfc_kont formula_wi
53
54  predicate wf_cnfc_kont (phi: cnfc_kont) = match phi with
55    | KCnfc_Id → true
56    | KCnfc_OrLeft phi k →
57        wf_negations_of_literals phi ∧ wf_cnfc_kont k
58    | KCnfc_OrRight k phi →
59        wf_negations_of_literals phi ∧ wf_conjunctions_of_disjunctions phi ∧
60        wf_cnfc_kont k
61    | KCnfc_AndLeft phi k →
62        wf_negations_of_literals phi ∧ wf_cnfc_kont k
63    | KCnfc_AndRight k phi →
64        wf_negations_of_literals phi ∧ wf_conjunctions_of_disjunctions phi ∧
65        wf_cnfc_kont k
66    end
67
68  type wf_cnfc_kont = {
69    cnfc_k: cnfc_kont;
70  } invariant { wf_cnfc_kont cnfc_k }
71    by { cnfc_k = KCnfc_Id }
72
73  let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
74  = match phi with
75    | FNeg phi1 → impl_free_desf_cps phi1 (KImpl_Neg k phi1)
76    | FOr phi1 phi2 → impl_free_desf_cps phi1 (KImpl_OrLeft phi2 k)
77    | FAnd phi1 phi2 → impl_free_desf_cps phi1 (KImpl_AndLeft phi2 k)
78    | FImpl phi1 phi2 → impl_free_desf_cps phi1 (KImpl_ImplLeft phi2 k)
79    | FConst phi → impl_apply (FConst_wi phi) k
80    | FVar phi → impl_apply (FVar_wi phi) k
81  end
```

```
82   with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
83   = match k with
84     | KImpl_Id → phi
85     | KImpl_Neg k phi1 → impl_apply (FNeg_wi phi) k
86     | KImpl_OrLeft phi1 k → impl_free_desf_cps phi1 (KImpl_OrRight k phi)
87     | KImpl_OrRight k phi2 → impl_apply (FOr_wi phi2 phi) k
88     | KImpl_AndLeft phi1 k → impl_free_desf_cps phi1 (KImpl_AndRight k phi)
89     | KImpl_AndRight k phi2 → impl_apply (FAnd_wi phi2 phi) k
90     | KImpl_ImplLeft phi1 k → impl_free_desf_cps phi1 (KImpl_ImplRight k phi)
91     | KImpl_ImplRight k phi2→ impl_apply (FOr_wi (FNeg_wi phi2) phi) k
92   end
93
94   let rec impl_desf_main (phi:formula) : formula_wi
95   = impl_free_desf_cps phi KImpl_Id
96
97   let rec nnfc_desf_cps (phi: formula_wi) (k: nnfc_kont) : formula_wi
98   = match phi with
99     | FNeg_wi (FNeg_wi phi1) → nnfc_desf_cps phi1 (Knnfc_negneg k phi1)
100    | FNeg_wi (FAnd_wi phi1 phi2) → nnfc_desf_cps (FNeg_wi phi1) (
     Knnfc_negandleft phi2 k)
101    | FNeg_wi (FOr_wi phi1 phi2) → nnfc_desf_cps (FNeg_wi phi1) (
     Knnfc_negorleft phi2 k)
102    | FOr_wi phi1 phi2 → nnfc_desf_cps phi1 (Knnfc_orleft phi2 k)
103    | FAnd_wi phi1 phi2 → nnfc_desf_cps phi1 (Knnfc_andleft phi2 k)
104    | phi → nnfc_apply phi k
105  end
106
107  with nnfc_apply (phi: formula_wi) (k: nnfc_kont) : formula_wi
108  = match k with
109    | Knnfc_id → phi
110    | Knnfc_negneg k phi1 → nnfc_apply phi k
111    | Knnfc_negandleft phi1 k → nnfc_desf_cps (FNeg_wi phi1) (Knnfc_negandright
      k phi)
112    | Knnfc_negandright k phi2 → nnfc_apply (FOr_wi phi2 phi) k
113    | Knnfc_negorleft phi1 k → nnfc_desf_cps (FNeg_wi phi1) (Knnfc_negorright k
      phi)
114    | Knnfc_negorright k phi2 → nnfc_apply (FAnd_wi phi2 phi) k
115    | Knnfc_andleft phi1 k → nnfc_desf_cps phi1 (Knnfc_andright k phi)
116    | Knnfc_andright k phi2 → nnfc_apply (FAnd_wi phi2 phi) k
117    | Knnfc_orleft phi1 k → nnfc_desf_cps phi1 (Knnfc_orright k phi)
118    | Knnfc_orright k phi2 → nnfc_apply (FOr_wi phi2 phi) k
119  end
```

```
120 let nnfc_desf_main (phi: formula_wi) : formula_wi
121 = nnfc_desf_cps phi Knnfc_id
122
123 let rec distr_desf_cps (phi1 phi2: formula_wi) (k: wf_distr_kont) : formula_wi
124 = match phi1,phi2 with
125   | FAnd_wi phi11 phi12, phi2 →
126      distr_desf_cps phi11 phi2 { distr_k = KDistr_Left phi12 phi2 k.distr_k }
127   | phi1, FAnd_wi phi21 phi22 →
128      distr_desf_cps phi1 phi21 { distr_k = KDistr_Left phi1 phi22 k.distr_k }
129   | phi1,phi2 → distr_apply (FOr_wi phi1 phi2) k
130 end
131
132 with distr_apply (phi: formula_wi) (k: wf_distr_kont) : formula_wi
133 = match k.distr_k with
134   | KDistr_Id → phi
135   | KDistr_Left phi1 phi2 k →
136      distr_desf_cps phi1 phi2 { distr_k = KDistr_Right k phi }
137   | KDistr_Right k phi1 →
138      distr_apply (FAnd_wi phi1 phi) { distr_k = k }
139 end
140
141 let distr_desf_main (phi1 phi2: formula_wi) : formula_wi
142 = distr_desf_cps phi1 phi2 { distr_k = KDistr_Id }
143
144 let rec cnfc_desf_cps (phi: formula_wi) (k: wf_cnfc_kont) : formula_wi
145 = match phi with
146   | FOr_wi phi1 phi2 →
147      cnfc_desf_cps phi1 { cnfc_k = KCnfc_OrLeft phi2 k.cnfc_k }
148   | FAnd_wi phi1 phi2 →
149      cnfc_desf_cps phi1 { cnfc_k = KCnfc_AndLeft phi2 k.cnfc_k }
150   | phi → cnfc_apply phi k
151 end
152
153 with cnfc_apply (phi: formula_wi) (k: wf_cnfc_kont) : formula_wi
154 = match k.cnfc_k with
155   | KCnfc_Id → phi
156   | KCnfc_OrLeft phi1 k → cnfc_desf_cps phi1 { cnfc_k = KCnfc_OrRight k phi }
157   | KCnfc_OrRight k phi2 →
158      cnfc_apply (distr_desf_cps phi2 phi { distr_k = KDistr_Id })
159      { cnfc_k = k }
160   | KCnfc_AndLeft phi1 k → cnfc_desf_cps phi1 {cnfc_k = KCnfc_AndRight k phi}
161   | KCnfc_AndRight k phi2 → cnfc_apply (FAnd_wi phi2 phi) { cnfc_k = k }
162 end
```

```
163     let cnfc_desf_main (phi: formula_wi) : formula_wi
164     = cnfc_desf_cps phi { cnfc_k = KCnfc_Id }
165
166     let t (phi: formula) : formula_wi
167     = cnfc_desf_main ( nnfc_desf_main ( impl_desf_main phi))
168
169 end
```

A.7 Defunctionalized Proof

```
1  predicate nnfc_post (k: nnfc_kont) (phi result: formula_wi)
2  = match k with
3      | Knnfc_id → let x = phi in x = result
4      | Knnfc_negneg k phi1 → let neg = phi in nnfc_post k phi result
5      | Knnfc_negandleft phi1 k → let hl = phi in nnfc_post k (FOr_wi phi (nnfc (
       FNeg_wi phi1))) result
6      | Knnfc_negandright k phi2 → let hr = phi in nnfc_post k (FOr_wi phi2 hr)
       result
7      | Knnfc_negorleft phi1 k → let hl = phi in nnfc_post k (FAnd_wi phi (nnfc (
       FNeg_wi phi1))) result
8      | Knnfc_negorright k phi2 →let hr = phi in nnfc_post k (FAnd_wi phi2 hr)
       result
9      | Knnfc_andleft phi1 k → let hl = phi in nnfc_post k (FAnd_wi phi (nnfc
       phi1)) result
10     | Knnfc_andright k phi2 → let hr = phi in nnfc_post k (FAnd_wi phi2 hr)
       result
11     | Knnfc_orleft phi1 k → let hl = phi in nnfc_post k (FOr_wi phi (nnfc phi1)
       ) result
12     | Knnfc_orright k phi2 → let hr = phi in nnfc_post k (FOr_wi phi2 hr)
       result
13 end
14
15 predicate impl_post (k: impl_kont) (phi result: formula_wi)
16 = match k with
17     | KImpl_Id → let x = phi in x = result
18     | KImpl_Neg k phi1 → let neg = phi in impl_post k (FNeg_wi phi) result
19     | KImpl_OrLeft phi1 k → let hl = phi in impl_post k (FOr_wi phi (impl_free
       phi1)) result
20     | KImpl_OrRight k phi2 → let hr = phi in impl_post k (FOr_wi phi2 hr)
       result
21     | KImpl_AndLeft phi1 k → let hl = phi in impl_post k (FAnd_wi phi (
       impl_free phi1)) result
22     | KImpl_AndRight k phi2 → let hr = phi in impl_post k (FAnd_wi phi2 hr)
       result
```

```
23      | KImpl_ImplLeft phi1 k → let hl = phi in impl_post k (FOr_wi (FNeg_wi phi)
         (impl_free phi1)) result
24      | KImpl_ImplRight k phi2→ let hr = phi in impl_post k (FOr_wi (FNeg_wi phi2
        ) hr) result
25  end
26
27  predicate distr_post (k: distr_kont) (phi result: formula_wi)
28  = match k with
29      | KDistr_Id → let x = phi in x = result
30      | KDistr_Left phi1 phi2 k → let hl = phi in distr_post k (FAnd_wi hl (distr
         phi1 phi2)) result
31      | KDistr_Right k phi1 → let hr = phi in distr_post k (FAnd_wi phi1 hr)
        result
32  end
33
34  predicate cnfc_post (k: cnfc_kont) (phi result: formula_wi)
35  = match k with
36      | KCnfc_Id → let x = phi in x = result
37      | KCnfc_OrLeft phi1 k → let hl = phi in cnfc_post k (distr hl (cnfc phi1))
        result
38      | KCnfc_OrRight k phi2 → let hr = phi in cnfc_post k (distr phi2 hr) result
39      | KCnfc_AndLeft phi1 k → let hl = phi in cnfc_post k (FAnd_wi phi (cnfc
        phi1)) result
40      | KCnfc_AndRight k phi2 → let hr = phi in cnfc_post k (FAnd_wi phi2 hr)
        result
41  end
42
43  let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
44    diverges
45    ensures{impl_post k (impl_free phi) result}
46  = ...
47
48  with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
49    diverges
50    ensures{impl_post k phi result}
51  = ...
52
53  let rec impl_desf_main (phi:formula) : formula_wi
54    diverges
55    ensures{ forall v. eval v phi = eval_wi v result }
56  = ...
57
58  let rec nnfc_desf_cps (phi: formula_wi) (k: nnfc_kont) : formula_wi
59    diverges
60    ensures{ nnfc_post k (nnfc phi) result }
61  = ...
```

```
62
63 with nnfc_apply (phi: formula_wi) (k: nnfc_kont) : formula_wi
64   diverges
65   ensures{ nnfc_post k phi result }
66 = ...
67
68 let nnfc_desf_main (phi: formula_wi) : formula_wi
69   diverges
70   ensures {(forall v. eval_wi v phi = eval_wi v result) ∧
     wf_negations_of_literals result}
71 = ...
72
73 let rec distr_desf_cps (phi1 phi2: formula_wi) (k: wf_distr_kont) : formula_wi
74   requires{ wf_negations_of_literals phi1 ∧ wf_negations_of_literals phi2}
75   requires{ wf_conjunctions_of_disjunctions phi1 ∧
     wf_conjunctions_of_disjunctions phi2}
76   diverges
77   ensures{ distr_post k.distr_k (distr phi1 phi2) result }
78 = ...
79
80 with distr_apply (phi: formula_wi) (k: wf_distr_kont) : formula_wi
81   requires{ wf_negations_of_literals phi}
82   requires{ wf_conjunctions_of_disjunctions phi}
83   diverges
84 ensures{ distr_post k.distr_k phi result }
85 = ...
86
87 let distr_desf_main (phi1 phi2: formula_wi) : formula_wi
88   diverges
89   requires{ wf_negations_of_literals phi1 ∧ wf_negations_of_literals phi2}
90   requires{ wf_conjunctions_of_disjunctions phi1 ∧
     wf_conjunctions_of_disjunctions phi2}
91   ensures { (forall v. eval_wi v (FOr_wi phi1 phi2) = eval_wi v result) } (*
     EVAL *)
92   ensures { wf_negations_of_literals result ∧ wf_conjunctions_of_disjunctions
     result }
93 = ...
94
95 let rec cnfc_desf_cps (phi: formula_wi) (k: wf_cnfc_kont) : formula_wi
96   requires{ wf_negations_of_literals phi }
97   diverges
98   ensures{ cnfc_post k.cnfc_k (cnfc phi) result }
99 = ...
```

```
100 with cnfc_apply (phi: formula_wi) (k: wf_cnfc_kont) : formula_wi
101   requires{ wf_negations_of_literals phi }
102   requires { wf_conjunctions_of_disjunctions phi }
103   diverges
104   ensures{ cnfc_post k.cnfc_k phi result }
105 = ...
106
107 let cnfc_desf_main (phi: formula_wi) : formula_wi
108   diverges
109   requires{ wf_negations_of_literals phi }
110   ensures{ (forall v. eval_wi v phi = eval_wi v result) ∧
        wf_negations_of_literals result}
111   ensures{ wf_conjunctions_of_disjunctions result}
112 = ...
113
114 let t (phi: formula) : formula_wi
115   diverges
116   ensures{(forall v. eval v phi = eval_wi v result) ∧ wf_negations_of_literals
        result ∧ wf_conjunctions_of_disjunctions result}
117 = ...
118
119 end
```

APPENDIX

B

Appendix 2 Hornify

B.1 Full Implementation

```
module Types

  type ident

  (* Conjunctive Normal Form *)

  type pliteral =
      | LBottom
      | LVar ident

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

  type formula_cnf =
      | FClause_cnf clause_cnf
      | FAnd_cnf formula_cnf formula_cnf


  (* Horn Formula *)

  type rightside =
      | RBottom
      | RTop
      | RVar ident
```

```
27
28    type positive =
29        | PLCBottom
30        | PLCVar ident
31        | PLCAnd positive positive
32
33    type leftside =
34        | LTop
35        | LPos positive
36
37    type basichornclause =
38        | BImpl leftside rightside
39
40    type hornclause =
41        | HBasic basichornclause
42        | HAnd hornclause hornclause
43
44  end
45
46
47  module Hornify
48
49    use option.Option, int.Int, bool.Bool, Types, Valuation, Lemmas
50
51    exception MoreThanOnePositive
52
53    let convertLiteralToR (pl: pliteral) : (rightside)
54    = match pl with
55        | LBottom → RBottom
56        | LVar x → RVar x
57      end
58
59    let convertLiteralToPLC (pl: pliteral) : (positive)
60    = match pl with
61        | LBottom → PLCBottom
62        | LVar x → PLCVar x
63      end
64
65    let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) :
         (rs: set, rp: option rightside)
66    = match pl with
67        | LBottom → let rbottom = Some (RBottom) in
68                    match nl with
69                      | LBottom → ((add (PLCBottom) s), rbottom)
70                      | LVar x → ((add (PLCVar x) s), rbottom)
71                    end
```

```
72        | LVar x → let rvar = Some (RVar x) in
73                      match nl with
74                        | LBottom → ((add (PLCBottom) s), rvar)
75                        | LVar x → ((add (PLCVar x) s), rvar)
76
77                      end
78     end
79
80    let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option
      rightside) : (rs: set, rp: option rightside)
81      raises{ MoreThanOnePositive }
82    = match p with
83        | None → addLiterals pl nl s p
84        | Some _ → raise MoreThanOnePositive
85      end
86
87
88    let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs:
      set, rp: option rightside)
89      raises{ MoreThanOnePositive }
90    variant{ phi }
91    = match phi with
92        | DOr_cnf (DLiteral _) (DLiteral _) → raise MoreThanOnePositive
93        | DOr_cnf (DLiteral pl) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (DLiteral pl
      ) → (* OR of positive literal and negative literal *)
94                                processCombination pl nl s p
95        | DOr_cnf (DNeg_cnf nl1) (DNeg_cnf nl2) → ((add (convertLiteralToPLC nl1)
       (add (convertLiteralToPLC nl2) s)), p)
96        | DOr_cnf (DOr_cnf phi1 phi2) (DLiteral pl) | DOr_cnf (DLiteral pl) (
      DOr_cnf phi1 phi2) → (* OR of binary or and positive literal *)
97                                match p with
98                                  | None → hornify_aux (DOr_cnf phi1 phi2) s (
      Some (convertLiteralToR pl))
99                                  | Some _ → raise MoreThanOnePositive
100                               end
101       | DOr_cnf (DOr_cnf phi1 phi2) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (
      DOr_cnf phi1 phi2) → (* Combination of binary or and negative literal *)
102                              hornify_aux (DOr_cnf phi1 phi2) (add (
      convertLiteralToPLC nl) s) p
103       | DOr_cnf phi1 phi2 → let (s1,p1) = hornify_aux phi1 s p in
104                               hornify_aux phi2 s1 p1
105       | _ → absurd
106     end
107
108
109   let conjunction (s: set): leftside
```

```
110   = let rec build (s: set)
111      = if(is_empty s) then absurd else
112        if((cardinal s) = 1) then (choose s) else
113        PLCAnd (choose s) (build (remove (choose s) s)) in
114     LPos (build s)
115
116
117   let getPositive (p: option rightside) : rightside
118   = match p with
119       | None → RBottom
120       | Some x → x
121     end
122
123   let getBasicHorn (phi: clause_cnf) : basichornclause
124     raises{ MoreThanOnePositive }
125   = match phi with
126       | DLiteral (LVar x) → BImpl (LTop) (RVar x)
127       | DLiteral (LBottom) → BImpl (LTop) (RBottom)
128       | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RBottom)
129       | DNeg_cnf (LBottom) → BImpl (LTop) (RTop)
130       | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
131                         BImpl (conjunction s) (getPositive p)
132     end
133
134   let rec hornify (phi: formula_cnf) : hornclause
135     raises{ MoreThanOnePositive }
136   = match phi with
137       | FClause_cnf phi1 → HBasic (getBasicHorn phi1)
138       | FAnd_cnf phi1 phi2 → HAnd (hornify phi1) (hornify phi2)
139   end
140
141 end
```

B.2 Evaluation Functions

```
1  module Valuation
2
3    use option.Option, int.Int, bool.Bool, Types
4
5    clone export set.SetApp with type elt = positive
6
7    use import set.Fset as FS
8
9    type valuation = ident → bool
10
```

```
11
12  (* CNF Valuation *)
13
14  let function eval_literal (v: valuation) (l: pliteral) : bool
15  = match l with
16      | LBottom → false
17      | LVar x → v x
18    end
19
20  let rec function eval_disj_cnf (v: valuation) (phi: clause_cnf) : bool
21  = match phi with
22      | DLiteral l → eval_literal v l
23      | DNeg_cnf l → not eval_literal v l
24      | DOr_cnf phi1 phi2 → eval_disj_cnf v phi1 || eval_disj_cnf v phi2
25    end
26
27  let rec function eval_formula_cnf (v: valuation) (phi: formula_cnf) : bool
28  = match phi with
29      | FClause_cnf phi1 → eval_disj_cnf v phi1
30      | FAnd_cnf phi1 phi2 → eval_formula_cnf v phi1 && eval_formula_cnf v phi2
31    end
32
33
34  (* Horn Valuation *)
35
36  let function eval_rightside (v: valuation) (r: rightside) : bool
37  = match r with
38      | RBottom → false
39      | RTop → true
40      | RVar x → v x
41    end
42
43  let rec function eval_positive (v: valuation) (plc: positive) : bool
44  = match plc with
45      | PLCBottom → false
46      | PLCVar x → v x
47      | PLCAnd phi1 phi2 → eval_positive v phi1 && (eval_positive v phi2)
48    end
49
50  let rec function eval_leftside (v: valuation) (l: leftside) : bool
51  = match l with
52      | LTop → true
53      | LPos phi1 → eval_positive v phi1
54    end
55
```

```
56  let rec function eval_basichornclause (v: valuation) (b: basichornclause) :
      bool
57  = match b with
58      | BImpl left right → implb (eval_leftside v left) (eval_rightside v right
      )
59    end
60
61  let rec function eval_hornclause (v: valuation) (h: hornclause) : bool
62  = match h with
63      | HBasic h1 → eval_basichornclause v h1
64      | HAnd h1 h2 → eval_hornclause v h1 && eval_hornclause v h2
65    end
66
67
68  (* Sets e Option Valuation *)
69
70  function eval_optionrightside (v: valuation) (p: option rightside) : bool
71  = match p with
72      | None → false
73      | Some x → eval_rightside v x
74    end
75
76  let rec ghost function eval_set (v: valuation) (s: fset positive) : bool
77  variant{FS.cardinal s}
78  = if FS.is_empty s then
79      false
80    else
81      if FS.cardinal s = 1 then
82        not (eval_positive v (FS.pick s))
83      else
84        let x = FS.pick s in
85           not (eval_positive v x) || eval_set v (FS.remove x s)
86
87  let rec ghost function eval_conjunction_set (v: valuation) (s: fset positive)
      : bool
88  variant{FS.cardinal s}
89  = if(FS.is_empty s) then
90      false
91    else
92      if FS.cardinal s = 1 then
93        eval_positive v (FS.pick s)
94      else
95        let x = FS.pick s in
96          eval_positive v x && eval_conjunction_set v (FS.remove x s)
97
98  (* Domain and Codomain Valuation *)
```

```
99
100   function eval_domain (v: valuation) (phi: clause_cnf) (s: set) (p: option
        rightside) : bool
101   = eval_disj_cnf v phi || eval_set v s || eval_optionrightside v p
102
103   function eval_codomain (v: valuation) (s: set) (p: option rightside) : bool
104   = eval_set v s || eval_optionrightside v p
105
106 end
```

B.3 Hornify Proof

```
1  module Hornify
2
3    use option.Option, int.Int, bool.Bool, Types, Valuation, Lemmas
4
5    exception MoreThanOnePositive
6
7    (* Functions *)
8
9    let convertLiteralToR (pl: pliteral) : (rightside)
10     ensures{ forall v. eval_literal v pl = eval_rightside v result }
11   = ..
12
13   let convertLiteralToPLC (pl: pliteral) : (positive)
14     ensures{ forall v. eval_literal v pl = eval_positive v result }
15   = ..
16
17   let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) :
        (rs: set, rp: option rightside)
18     requires{ p = None }
19     ensures{ (not is_empty rs) }
20     ensures { forall v. (eval_literal v pl || (not (eval_literal v nl)) ||
       eval_set v s || eval_optionrightside v p) = (eval_set v rs ||
       eval_optionrightside v rp) }
21   = ...
22
23   let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option
       rightside) : (rs: set, rp: option rightside)
24     raises{ MoreThanOnePositive }
25     ensures{ (not is_empty rs) }
26     ensures { forall v. (eval_literal v pl || not (eval_literal v nl) ||
       eval_set v s || eval_optionrightside v p) = (eval_set v rs ||
       eval_optionrightside v rp) }
27   = ...
```

```
28
29
30    let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs:
        set, rp: option rightside)
31      requires{ exists x y. phi = DOr_cnf x y }
32      ensures{ (not is_empty rs) }
33      ensures{ forall v. eval_domain v phi s p = eval_codomain v rs rp }
34      ensures{ forall v. eval_domain v phi s p = implb (eval_conjunction_set v rs)
         (eval_optionrightside v rp) }
35      raises{ MoreThanOnePositive }
36      variant{ phi }
37    = ...
38
39
40    let conjunction (s: set): leftside
41      requires{not is_empty s}
42      ensures{forall v. eval_conjunction_set v s = eval_leftside v result }
43    = let rec build (s: set)
44       requires{not is_empty s}
45       ensures{forall v. eval_conjunction_set v s = eval_positive v result}
46       variant{cardinal s}
47       = ...
48
49
50    let getPositive (p: option rightside) : rightside
51      ensures{forall v. eval_optionrightside v p = eval_rightside v result}
52    = ...
53
54    let getBasicHorn (phi: clause_cnf) : basichornclause
55      ensures{ forall v. eval_disj_cnf v phi = eval_basichornclause v result }
56      raises{ MoreThanOnePositive }
57    = ...
58
59    let rec hornify (phi: formula_cnf) : hornclause
60      raises{ MoreThanOnePositive }
61      ensures{forall v. eval_formula_cnf v phi = eval_hornclause v result}
62      variant{ phi }
63    = ...
64
65
66  end
```