



**Pedro Miguel Laforet Barroso**

Bachelor Degree

## **Formally Verified Bug-free Implementations of (Logical) Algorithms**

Relatório intermédio 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

February, 2019



## ACKNOWLEDGEMENTS

This intermediate report was supported by the Tezos Foundation.

## 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 uma 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

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Software Bugs and Vulnerabilities . . . . .	1
1.2	Testing and Program Verification . . . . .	1
1.3	FACTOR . . . . .	2
1.4	Document Structure . . . . .	2
<b>2</b>	<b>Problem</b>	<b>3</b>
2.1	Computational Logic courses are either too informal or too formal . . . . .	3
2.2	Lack of material about correction . . . . .	3
<b>3</b>	<b>Objectives and Contributions</b>	<b>5</b>
3.1	Objectives . . . . .	5
3.2	Contributions . . . . .	6
3.3	Implementation and Verification of T Algorithm . . . . .	6
3.3.1	Implementation . . . . .	8
3.3.2	Correctness Criteria . . . . .	10
3.3.3	Verification . . . . .	11
3.3.4	Conclusions and Observations . . . . .	16
<b>4</b>	<b>State of the Art</b>	<b>17</b>
4.1	Computational Logic . . . . .	17
4.1.1	Structure . . . . .	17
4.1.2	Global overview of Computational Logic in Portugal . . . . .	18
4.2	Tools to support Computational Logic . . . . .	20
4.2.1	Tarski's World . . . . .	20
4.2.2	Fitch . . . . .	21
4.2.3	Boole . . . . .	22
4.3	Proof-Assistant Tools . . . . .	22
4.3.1	Why3 . . . . .	22
4.3.2	Dafny . . . . .	22
4.3.3	Coq . . . . .	23
4.3.4	Isabelle/HoL . . . . .	23
4.3.5	Twelf . . . . .	23



CONTENTS

---

4.3.6	Agda . . . . .	23
4.3.7	CFML . . . . .	23
<b>5</b>	<b>Plan and Schedule</b>	<b>25</b>
5.1	Plan . . . . .	25
5.2	Schedule . . . . .	25
	<b>Bibliography</b>	<b>27</b>

## INTRODUÇÃO

### 1.1 Software Bugs and Vulnerabilities

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 [15], 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 [22].

### 1.2 Testing and Program Verification

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

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. Formany, there are no methodologies or tools for verification on daily basis of programming.

The process of formal verification 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 [14], so named because it is made up of three components: pre-condition, program and post-condition.

$$(pre - condition) P (post - condition)$$

The triple means that, if the program  $P$  is run and the program pre-condition are satisfied, then the resulting state will satisfy the post-conditions.

### 1.3 FACTOR

This dissertation is part of the FACTOR (Functional ApproaCh Teaching pOrtuguese couRses) project [11]. FACTOR is funded by the Tezos Foundation [20] 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.

### 1.4 Document Structure

This report has been organised as follows: Chapter 2 describes the problems in the Computational Logic unit and Chapter 3 proposes a solution to these problems. Chapter 4 reports the aspects of Computational Logic unit, the existing tools to support this unit and the main proof-assistant tools. Chapter 5 defines the plan and its schedule.

## PROBLEM

## 2.1 Computational Logic courses are either too informal or too formal

Computational Logic 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. In 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 will be later discussed in Section 4.1.

## 2.2 Lack of material about correction

As mention in section 1.2, 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.

## OBJECTIVES AND CONTRIBUTIONS

### 3.1 Objectives

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 [19].

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 4.3.

## 3.2 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.

To demonstrate that the objective is possible and also as preparation for this dissertation was suggested to implement and verify an algorithm that converts propositional formulas in their conjunctive normal form. The next section presents such implementation and posterior verification.

## 3.3 Implementation and Verification of T Algorithm

The algorithm takes a formula in the set of propositional formulas ( $G_p$ ) obtained considering the primitive negation connective ( $\neg$ ) and defining the constant  $\perp$  as abbreviation:  $\perp \stackrel{abv}{=} \phi \wedge \neg\phi$ , where  $\phi \in G_p$  and transforms it into a formula in the set of propositional formulas ( $H_p$ , where  $H_p \subseteq G_p$ ) that does not contain the implication connective ( $\rightarrow$ ).

**Definition:** let  $T : G_p \rightarrow H_p$  be the following function:

$$T(\phi) = CNFC(NNFC(ImplFree(\phi))) \quad (3.1)$$

**Theorem:** The algorithm is correct if given  $\phi \in G_p$ , it yields  $\psi \in H_p$  with  $\psi = T(\phi)$ , where  $\psi$  is in conjunctive normal form and  $\phi \equiv \psi$ .

The algorithm, as seen in its definition, is divided into three steps:

1. ImplFree - that removes the implication connective ( $G_p \rightarrow H_p$ ).
2. NNFC - that removes double negations ( $\neg\neg$ ).
3. CNFC - that converts a proposition formula without the implication connective and double negations to its conjunctive normal form.

Here are the definitions of each step.

**ImplFree:**

**Definition:** let  $ImplFree : G_p \rightarrow H_p$  be the following recursive function:

$$ImplFree(\varphi) = \begin{cases} \neg ImplFree(\varphi_1), & \text{if } \varphi = \neg \varphi_1 \\ ImplFree(\varphi_1) \vee ImplFree(\varphi_2), & \text{if } \varphi = \varphi_1 \vee \varphi_2 \\ ImplFree(\varphi_1) \wedge ImplFree(\varphi_2), & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \\ \neg ImplFree(\varphi_1) \vee ImplFree(\varphi_2), & \text{if } \varphi = \varphi_1 \rightarrow \varphi_2 \\ \varphi, & \text{otherwise} \end{cases} \quad (3.2)$$

Note that  $\varphi_1 \rightarrow \varphi_2 = \neg \varphi_1 \vee \varphi_2$  and that the base case of this function is when the formula  $\varphi$  is a propositional symbol.

**NNFC:**

**Definition:** let  $NNFC : H_p \rightarrow H_p$  be the following recursive function:

$$NNFC(\varphi) = \begin{cases} NNFC(\varphi_1), & \text{if } \varphi = \neg \neg \varphi_1 \\ NNFC(\neg \varphi_1) \vee NNFC(\neg \varphi_2), & \text{if } \varphi = \neg(\varphi_1 \wedge \varphi_2) \\ NNFC(\neg \varphi_1) \wedge NNFC(\neg \varphi_2), & \text{if } \varphi = \neg(\varphi_1 \vee \varphi_2) \\ NNFC(\varphi_1) \vee NNFC(\varphi_2), & \text{if } \varphi = \varphi_1 \vee \varphi_2 \\ NNFC(\varphi_1) \wedge NNFC(\varphi_2), & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \\ \varphi, & \text{otherwise} \end{cases} \quad (3.3)$$

Note that the base case of this function is when the formula  $\varphi$  is a propositional symbol or its negation.

**CNFC:**

**Definition:** let  $CNFC : H_p \rightarrow H_p$  be the following recursive function:

$$CNFC(\varphi) = \begin{cases} Distr(CNFC(\varphi_1), CNFC(\varphi_2)), & \text{if } \varphi = \varphi_1 \vee \varphi_2 \\ CNFC(\varphi_1) \wedge CNFC(\varphi_2), & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \\ \varphi, & \text{otherwise} \end{cases} \quad (3.4)$$

and being  $CNFC : H_p \times H_p \rightarrow H_p$  the following function:

$$Distr(\varphi_1, \varphi_2) = \begin{cases} Distr(\varphi_{11}, \varphi_2) \wedge Distr(\varphi_{12}, \varphi_2), & \text{if } \varphi_1 = \varphi_{11} \wedge \varphi_{12} \\ Distr(\varphi_1, \varphi_{21}) \wedge Distr(\varphi_1, \varphi_{22}), & \text{if } \varphi_2 = \varphi_{21} \wedge \varphi_{22} \\ \varphi_1 \vee \varphi_2, & \text{otherwise} \end{cases} \quad (3.5)$$



### 3.3.1 Implementation

The implementation was developed using WhyML. However, for a clear demonstration was decided to show OCaml code instead of WhyML. The OCaml code presented was extracted from WhyML using the automated extraction feature of Why3.

Considering that the implementation is under a functional language, the implementation is merely trivial and easily related to their functions definitions described in Section 3.3.

The least trivial step may be the definition of the type formula. A formula can either be:

1. a Variable (FVar) (e.g.  $x, y, z$ )
2. a Constant (FConst) (e.g.  $\text{true}, \text{false}$ )
3. an And (FAnd) between two formulas (e.g.  $x \wedge y$ )
4. an Or (FOr) between two formulas (e.g.  $x \vee y$ )
5. an Implication (FImpl) between two formulas (e.g.  $x \rightarrow y$ )
6. a Negation (FNeg) of a formula (e.g.  $\neg x$ )

Although, this is relatively easy to implement with the benefits of symbolic manipulation that functional languages provides.

```
1 type formula =  
2   | FVar of ident  
3   | FConst of bool  
4   | FAnd of formula * formula  
5   | FOr of formula * formula  
6   | FImpl of formula * formula  
7   | FNeg of formula
```

Listing 3.1: OCaml Type Formula

With the type formula defined the implementation of the functions is almost a copy the mathematical definitions into OCaml, below is presented each function code.

```
1 let t (phi: formula) : formula = cnfc (nnfc (impl_free phi))  
2
```

Listing 3.2: T OCaml Function

```
1 let rec impl_free (phi: formula) : formula =  
2   begin match phi with  
3   | FNeg phi1 → FNeg (impl_free phi1)
```

```

4 | FOr (phi1, phi2) → FOr ((impl_free phi1), (impl_free phi2))
5 | FAnd (phi1, phi2) → FAnd ((impl_free phi1), (impl_free phi2))
6 | FImpl (phi1, phi2) → FOr ((FNeg (impl_free phi1)), (impl_free phi2))
7 | FConst phi1 → FConst phi1
8 | FVar phi1 → FVar phi1
9 end
10

```

Listing 3.3: ImplFree OCaml Function

```

1 let rec nnfc (phi: formula) : formula =
2   begin match phi with
3   | FNeg FNeg phi1 → nnfc phi1
4   | FNeg FAnd (phi1, phi2) → FOr ((nnfc (FNeg phi1)), (nnfc (FNeg phi2)))
5   | FNeg FOr (phi1, phi2) → FAnd ((nnfc (FNeg phi1)), (nnfc (FNeg phi2)))
6   | FOr (phi1, phi2) → FOr ((nnfc phi1), (nnfc phi2))
7   | FAnd (phi1, phi2) → FAnd ((nnfc phi1), (nnfc phi2))
8   | phi1 → phi1
9   end
10

```

Listing 3.4: NNFC OCaml Function

```

1 let rec cnfc (phi: formula) : formula =
2   begin match phi with
3   | FOr (phi1, phi2) → let o = cnfc phi2 in let o1 = cnfc phi1 in distr o1 o
4   | FAnd (phi1, phi2) →
5     let o = cnfc phi2 in let o1 = cnfc phi1 in FAnd (o1, o)
6   | phi1 → phi1
7   end

```

Listing 3.5: CNFC OCaml Function

```

1 let rec distr (phi1: formula) (phi2: formula) : formula =
2   begin match (phi1, phi2) with
3   | (FAnd (phi11, phi12), phi21) →
4     let o = distr phi12 phi21 in let o1 = distr phi11 phi21 in FAnd (o1, o)
5   | (phi11, FAnd (phi21, phi22)) →
6     let o = distr phi11 phi22 in let o1 = distr phi11 phi21 in FAnd (o1, o)
7   | (phi11, phi21) → FOr (phi11, phi21)
8   end

```

Listing 3.6: Distr OCaml Function

### 3.3.2 Correctness Criteria

The first thing to do when we want to verify an algorithm is to define correctness criteria according to its specification. According to the T algorithm theorem 3.1, the algorithm is correct if given  $\phi \in G_p$ , there is possible to get  $\psi \in H_p$  doing  $\psi = T(\phi)$ , where  $\psi$  is in conjunctive normal form and  $\phi \equiv \psi$ .

From this, we can already define the correctness criteria of the algorithm:

1. The result must have no implication connectives.
2. The result must be equivalent to the original formula.
3. The result must be in a conjunctive normal form.

The same was made for each step of the algorithm, defining each correctness criteria.

#### **ImplFree:**

1. The result must have no implication connectives.
2. The result must be equivalent to the original formula.

#### **NNFC:**

1. The result must have no implication connectives.
2. The result must have no double negations and the negations should be of literals, i.e., variables or constants.
3. The result must be equivalent to the original formula.

#### **CNFC and Distr:**

1. The result must have no implication connectives.
2. The result must have no double negations and the negations should be of literals, i.e., variables or constants.
3. The result must be in conjunctive normal form.
4. The result must be equivalent to the original formula.

### 3.3.3 Verification

As seen in Section 3.3.1 the implementation of these type of algorithms is easy. Although, verification can rise some difficulties. In this section, the proof of the T Algorithm will be presented **according** to correctness criteria defined in Section 3.3.2. The proof of each criteria will be presented below.

**The result must be equivalent to the original formula:**

A valuation type was added, in order to map each variable to a Boolean value.

```
1 type valuation = ident → bool
```

This type will serve as a base case for the evaluation function. The evaluation function receives the valuation, that contains the value of each variable, and the formula to be evaluated. This function follows the simple rules of logic evaluation:

- If it is a *variable*, the evaluation is the value associated with that *variable*.
- If it is a *constant*, the evaluation is the value of the *constant*.
- If it is a *negation*, the evaluation is the opposite of the value of the formula that is being negated.
- If it is an *and* it evaluates both arguments of the *and*. In order to the *and* to be true, both arguments need to be true.
- If it is an *or* it evaluates each argument of the *or*. In order to the *or* to be true, one of the arguments need to be true.
- If it is an *implication* and since  $A \rightarrow B = \neg A \vee B$ , the evaluation is made as an standard *or*.

```
1 function eval (v: valuation) (f: formula) : bool
2   = match f with
3     | FVar x      → v x
4     | FConst b    → b
5     | FAnd f1 f2  → eval v f1 && eval v f2
6     | FOr f1 f2   → eval v f1 || eval v f2
7     | FImpl f1 f2 → not (eval v f1) || eval v f2
8     | FNeg f      → not (eval v f)
9 end
```

Listing 3.7: Eval function

**The result must have no implication connectives:**

An additional definition of formula (formula\_wi) and eval function was made, but this time without the implication connective. Since this type has no implication connective by definition it assures that if a formula is of type formula\_wi then it has no implication connectives. This is called a pre-syntactic condition.

```
1 type formula_wi =
2   | FVar_wi ident
3   | FConst_wi bool
4   | FAnd_wi formula_wi formula_wi
5   | FOr_wi formula_wi formula_wi
6   | FNeg_wi formula_wi
7
8 function eval_wi (v: valuation) (f: formula_wi) : bool
9   = match f with
10    | FVar_wi x      → v x
11    | FConst_wi b    → b
12    | FAnd_wi f1 f2  → eval_wi v f1 && eval_wi v f2
13    | FOr_wi f1 f2   → eval_wi v f1 || eval_wi v f2
14    | FNeg_wi f      → not (eval_wi v f)
15 end
```

The proof is done by simply converting the formula type to the formula\_wi type and ensuring that both original and returning formula are equivalent, i.e., have the same evaluation.

```
1 let rec impl_free (phi: formula) : formula_wi
2   variant { phi }
3   ensures { forall v. eval v phi = eval_wi v result }
4   = match phi with
5   | FNeg phi1 → FNeg_wi (impl_free phi1)
6   | FOr phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
7   | FAnd phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
8   | FImpl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
9   | FConst phi → FConst_wi phi
10  | FVar phi → FVar_wi phi
11 end
```

### The result must have no double negations and the negations should be of literals:

A well-formation predicate that checks if the negation is close to a literal was created to assure these criteria. To assure that negation is close to a literal we need to check each case:

- If it is a *negation*, we need to check that the formula that is being negated is not an *and*, *or* or another *negation* and then recursively check the formula that is being negated.

- If it is an *or* or an *and*, we need to check that both arguments have only negation of literals.
- If it is a *variable* or *constant*, then we reach to the base case and hence return true.

```

1 predicate wf_negations_of_literals (f: formula_wi)
2   = match f with
3     | FNeg_wi f → (forall f1 f2. f ≠ FOr_wi f1 f2 ∧ f ≠ FAnd_wi f1 f2 ∧ f ≠
4       FNeg_wi f1) ∧ wf_negations_of_literals f
5     | FOr_wi f1 f2 | FAnd_wi f1 f2 → wf_negations_of_literals f1 ∧
6       wf_negations_of_literals f2
7     | FVar_wi _ → true
8     | FConst_wi _ → true
9   end

```

Listing 3.8: Negations of literals well-formulated predicate

Then the NNFC function must ensure this well-formation predicate and that the original formula has the same evaluation of the returning formula.

```

1 let rec nnfc (phi: formula_wi)
2   variant { size phi }
3   ensures { (forall v. eval_wi v phi = eval_wi v result) ∧
4     wf_negations_of_literals result }
5   = match phi with
6     | FNeg_wi (FNeg_wi phi1) → nnfc phi1
7     | FNeg_wi (FAnd_wi phi1 phi2) → FOr_wi (nnfc (FNeg_wi phi1)) (nnfc (FNeg_wi
8       phi2))
9     | FNeg_wi (FOr_wi phi1 phi2) → FAnd_wi (nnfc (FNeg_wi phi1)) (nnfc (FNeg_wi
10      phi2))
11    | FOr_wi phi1 phi2 → FOr_wi (nnfc phi1) (nnfc phi2)
12    | FAnd_wi phi1 phi2 → FAnd_wi (nnfc phi1) (nnfc phi2)
13    | phi → phi
14  end

```

Why3 could not prove termination, because there are cases of the recursive function where the function is being called with added connectives, so to provide a good variant and hence prove termination, a simple size function that counts the number of elements in a formula needs to be created.

```

1 function size (phi: formula_wi) : int = match phi with
2   | FVar_wi _ | FConst_wi _ → 1
3   | FNeg_wi phi → 1 + size phi
4   | FAnd_wi phi1 phi2 | FOr_wi phi1 phi2 →
5     1 + size phi1 + size phi2
6   end

```

However, even with this size function, it seems that the Why3 could not prove termination. This is because the Why3 does not have the information that the size of the formulas is always greater or equal to zero, so we need to explicit detail it in the code with an auxiliary lemma.

```
1 let rec lemma size_nonneg (phi: formula_wi)
2   variant { phi }
3   ensures { size phi ≥ 0 }
4   = match phi with
5     | FVar_wi _ | FConst_wi _ → ()
6     | FNeg_wi phi → size_nonneg phi
7     | FAnd_wi phi1 phi2 | FOr_wi phi1 phi2 →
8       size_nonneg phi1; size_nonneg phi2
9   end
```

**The result must be in a conjunctive normal form:**

Similar to the previous criteria, a well-formulated predicate was created to ensure that the result must be in a conjunctive normal form, i.e., that after an *or* we cannot find an *and*.

```
1 predicate wf_conjunctions_of_disjunctions (f: formula_wi)
2   = match f with
3     | FAnd_wi f1 f2 → wf_conjunctions_of_disjunctions f1 ∧
4       wf_conjunctions_of_disjunctions f2
5     | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
6     | FConst_wi _ → true
7     | FVar_wi _ → true
8     | FNeg_wi f1 → wf_conjunctions_of_disjunctions f1
9   end
```

The function is called recursively until an *or* is reached then we need to call another function that checks that there is no *and* thenceforward.

```
1 predicate wf_disjunctions (f: formula_wi)
2   = match f with
3     | FAnd_wi _ _ → false
4     | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
5     | FConst_wi _ → true
6     | FVar_wi _ → true
7     | FNeg_wi f1 → wf_disjunctions f1
8   end
```

Then the CNFC function must ensure the *wf\_conjunctions\_of\_disjunctions* well-formation predicate, still ensure that the previous one's predicates were not violated and that the original formula has the same evaluation of the returning formula.

```

1 let rec cnfc (phi: formula_wi)
2   requires{ wf_negations_of_literals phi }
3   variant { phi }
4   ensures{ (forall v. eval_wi v phi = eval_wi v result) ∧
   wf_negations_of_literals result}
5   ensures{ wf_conjunctions_of_disjunctions result}
6   = match phi with
7     | FOr_wi phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
8     | FAnd_wi phi1 phi2 → FAnd_wi (cnfc phi1) (cnfc phi2)
9     | phi → phi
10  end

```

To ensure the *wf\_negations\_of\_literals* predicate, we need to add this predicate to the pre-condition of the function.

```

1 let rec cnfc (phi: formula_wi)
2   requires{ wf_negations_of_literals phi }
3   variant { phi }
4   ensures{ (forall v. eval_wi v phi = eval_wi v result) ∧
   wf_negations_of_literals result}
5   ensures{ wf_conjunctions_of_disjunctions result}
6   = match phi with
7     | FOr_wi phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
8     | FAnd_wi phi1 phi2 → FAnd_wi (cnfc phi1) (cnfc phi2)
9     | phi → phi
10  end

```

Why3 only with this information, could not prove both predicates and evaluation. This because we need to ensure that the *distr* function also ensures the same predicates and evaluation conditions.

In order for *distr* to prove that an *or* of arguments that are not *and* (base case) is a conjunction of disjunction, we need to require that both arguments respects the predicate *wf\_conjunctions\_of\_disjunctions*.

```

1 let rec distr (phi1 phi2: formula_wi)
2   requires{ wf_negations_of_literals phi1 ∧ wf_negations_of_literals phi2}
3   requires{ wf_conjunctions_of_disjunctions phi1 ∧
   wf_conjunctions_of_disjunctions phi2}
4   variant { size phi1 + size phi2 }
5   ensures { (forall v. eval_wi v (FOr_wi phi1 phi2) = eval_wi v result) }
6   ensures { wf_negations_of_literals result ∧ wf_conjunctions_of_disjunctions
   result }
7   = match phi1, phi2 with
8     | FAnd_wi phi11 phi12, phi2 → FAnd_wi (distr phi11 phi2) (distr phi12 phi2)
9     | phi1, FAnd_wi phi21 phi22 → FAnd_wi (distr phi1 phi21) (distr phi1 phi22)
10    | phi1, phi2 → FOr_wi phi1 phi2

```



11 `end`

Why3 could not prove that an *or* of two conjunctions of disjunctions is a conjunction of disjunctions, that because although it is easy to a human to understand, for a machine is complex. An auxiliary lemma was provided in order to help the machine. This lemma states that if a specific formula is a conjunction of disjunctions, has only negations of literals and does not contain any *and* then it must be a disjunction.

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

### T Algorithm

Now that every single sub-function of the T Algorithm has been successfully proved, we simply ensure the correctness criteria of the algorithm adding each criteria to the post-conditions.

```
1 let t (phi: formula)  
2   ensures { (forall v. eval v phi = eval_wi v result) ∧  
3     wf_negations_of_literals result ∧ wf_conjunctions_of_disjunctions result}  
   = cnfc (nnfc (impl_free phi))
```

#### 3.3.4 Conclusions and Observations

The implementation of the algorithm took a few hours, while the verification process took two weeks. This demonstrates that although the implementation of propositional algorithms may be trivial, the process of verifying such implementations is not that simple.

After the verification process, the implementation was used to solve some exercises of propositional logic given in the University of Beira Interior that were previously "hand-solved". The implementation was really great to give the confidence in some resolutions and also to catch some mistakes due to lack of attention in other resolutions.

## STATE OF THE ART

### 4.1 Computational Logic

According to the 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science of ACM and IEEE [9] 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 [18].

Computational Logic is a mandatory unit worthing 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.

#### 4.1.1 Structure

The structure of Computational Logic is the following:

1. Propositional Logic
  - a) Syntax:
    - i. inductive definition of propositional language
    - ii. terms from descriptions in natural language
  - b) Semantics:
    - i. valuation and structure of interpretation: satisfaction rate

- ii. truth tables
    - iii. validity and semantic consequence; logical equivalence
  - c) Deductive system: natural deduction
    - i. notions of derivation and proof
    - ii. rules of introduction and elimination
    - iii. correctness and completeness of the system
  - d) Decision algorithms
    - i. conjunctive normal form and clausal form
    - ii. Horn's algorithm
    - iii. resolution
2. First-Order Logic
- a) Syntax:
    - i. inductive definition of language
    - ii. terms from descriptions in natural language
    - iii. concept of free variables and substitution
  - b) Semantics:
    - i. valuation and structure of interpretation: satisfaction rate
    - ii. validity and semantic consequence; logical equivalence
    - iii. axioms (including De Morgan's laws)
  - c) Deductive system: natural deduction
    - i. notions of derivation and proof
    - ii. rules of introduction and elimination
    - iii. correctness and completeness of the system
  - d) Decision algorithms
    - i. prenex normal form and Skolem normal form
    - ii. skolemization and unification
    - iii. resolution

#### 4.1.2 Global overview of Computational Logic in Portugal

*The overview here described is based on "Report on the Curricular Unit Computational Logic" of the professor António Ravara [18] 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 [16]. 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) [17]

#### **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 [12].

#### **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 [25].

#### **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 [18].

## 4.2 Tools to support Computational Logic

### 4.2.1 Tarski's World

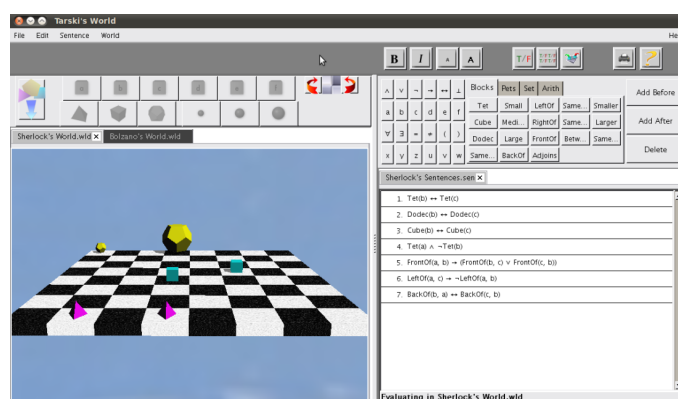


Figure 4.1: OpenProof (n.d.). Tarski's World main window. [image] Available at: <https://ggweb.gradegrinder.net/support/manual/tarski>.

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 [3]. 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 4.1 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.

### 4.2.2 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  $P \leftrightarrow \neg\neg P$ : (this is an example 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]

Listing 4.1:  $P \leftrightarrow \neg\neg P$  proof.

There is a tool called Fitch-checker that helps the construction and validation of this style of deduction. In the figure 4.3 is presented the proof of  $A \rightarrow B, A \rightarrow C \therefore A \rightarrow (B \wedge C)$  with this tool.

Construct a proof for the argument:  $A \rightarrow B, A \rightarrow C \therefore A \rightarrow (B \wedge C)$

1 |  $A \rightarrow B$

2 |  $A \rightarrow C$

3 |  $A$

4 |  $B$   $\rightarrow E$  1, 3

5 |  $C$   $\rightarrow E$  2, 3

6 |  $B \wedge C$   $\wedge I$  4, 5

7 |  $A \rightarrow (B \wedge C)$   $\rightarrow I$  3-6

NEW LINE NEW SUBPROOF

🎉 Congratulations! This proof is correct.

CHECK PROOF START OVER

Figure 4.2: Fitch-checker  $A \rightarrow B, A \rightarrow C \therefore A \rightarrow (B \wedge C)$  proof. [image] Available at: <https://github.com/OpenLogicProject/fitch-checker>.

### 4.2.3 Boole

Boole [5] is a simple tool that helps students to build and verify truth tables.

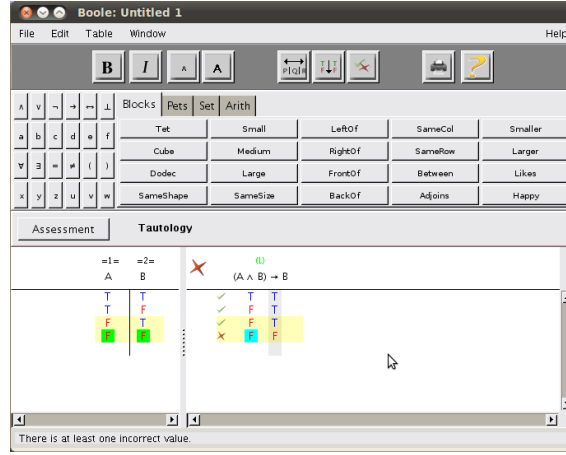


Figure 4.3: Fitch-checker  $A \rightarrow B, A \rightarrow C \therefore A \rightarrow (B \wedge C)$  proof. [image] Available at: <https://github.com/OpenLogicProject/fitch-checker>.

## 4.3 Proof-Assistant Tools

Proof assistant tools are computer systems to assist with the development of formal proofs by human-machine collaboration [13]. In this section, a brief description of some tools will be presented.

### 4.3.1 Why3

Why3 is a platform for deductive program verification [4]. 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 [4]. 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.

### 4.3.2 Dafny

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 [10]. Dafny is the closest tool to Why3 but has the downside that only supports the Z3 prover.

### 4.3.3 Coq

Coq 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 [21]. Similar to Why3, Coq can extract certified OCaml programs.

### 4.3.4 Isabelle/HoL

Isabelle/HOL 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 [8]. Similar to the other theorem provers described, Isabelle/HOL also has his special ML programming language.

### 4.3.5 Twelf

Twelf is a language used to specify, implement, and prove properties of deductive systems such as programming languages and logics [23]. The Twelf theorem proving component is at an experimental stage and currently under active development [24].

### 4.3.6 Agda

Agda 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 [2]. However, it has no support for tactics <sup>1</sup>.

### 4.3.7 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 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 [6].

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 [7].

---

<sup>1</sup>A tactic replaces the goal with the subgoals it generates.





## PLAN AND SCHEDULE

### 5.1 Plan

The plan is to finish the implementation and verification of the T algorithm described in Section 3.3, adding Continuation-passing style with undoes and then fully implement and verify other relevant algorithms manipulating propositional and even first-order formula like Horn algorithm and resolution algorithm. The algorithms will be subject to the following process:

1. Identification and analysis of properties and specifications of the algorithm.
2. Implement the algorithm in a functional language.
  - a) "Direct"implementation
  - b) CPS implementation
  - c) CPS with undoes implementation
3. Correctness proof of the implementation with regard to the envisaged properties using the Why3 program verification platform.

### 5.2 Schedule

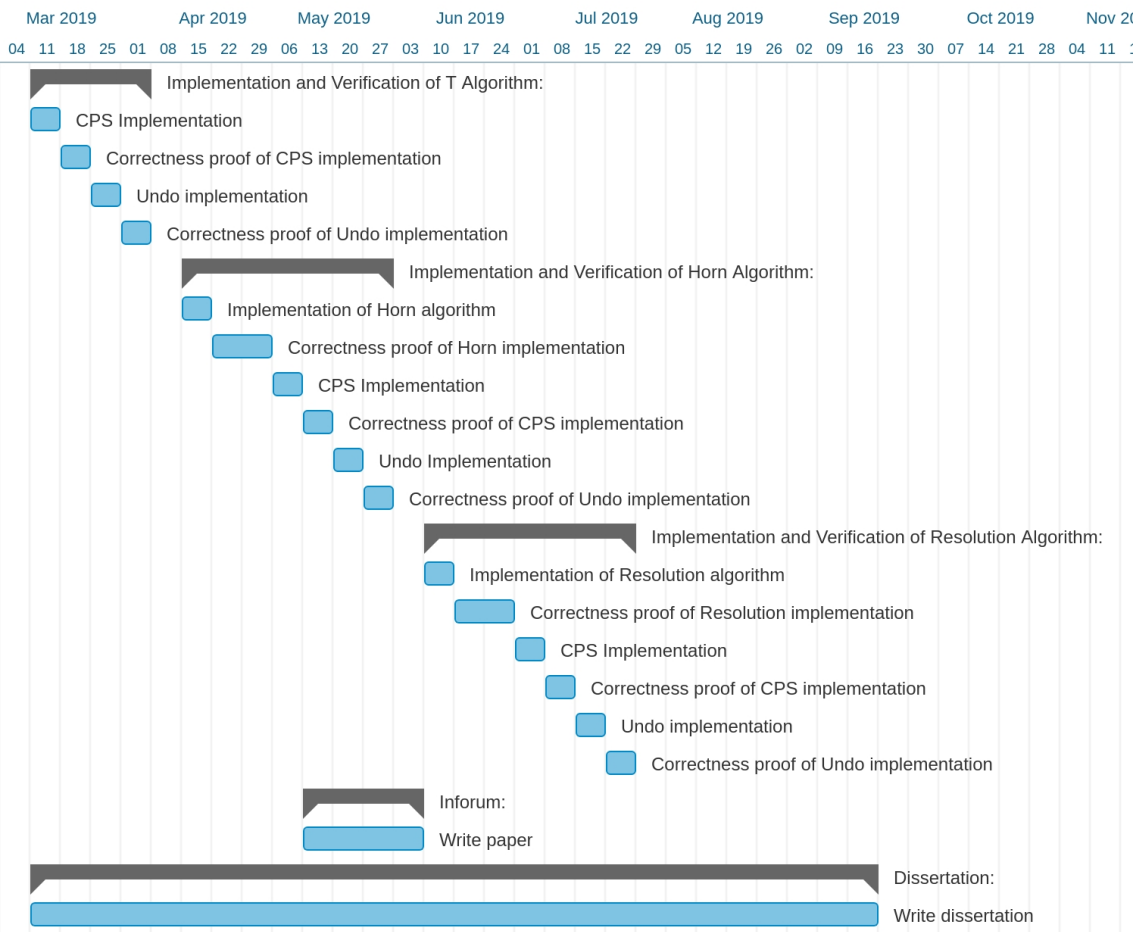


Figure 5.1: Schedule.

## BIBLIOGRAPHY

- [1] W. Ackermann. “Frederic Brenton Fitch. Symbolic logic. An introduction.” In: *The journal of symbolic logic* (1952).
- [2] Agda. URL: <https://wiki.portal.chalmers.se/agda/pmwiki.php>.
- [3] D. Barker-Plummer, J. Barwise, and J. Etchemendy. *Tarski’s World: Revised and Expanded*. Center for the Study of Language and Inf, 2007.
- [4] 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.
- [5] *Boole Manual*. URL: <https://ggweb.gradegrinder.net/support/manual/boole>.
- [6] A. Charguéraud. “Characteristic formulae for the verification of imperative programs.” In: *ACM SIGPLAN Notices*. ACM. 2011.
- [7] A. Charguéraud. URL: <http://www.chargueraud.org/softs/cfml/>.
- [8] 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).
- [9] 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](https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf).
- [10] *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/>.
- [11] *FACTOR: Functional Approach Teaching Portuguese courses*. URL: <http://ctp.di.fct.unl.pt/FACTOR/>.
- [12] FenixEdu. *Logic for Programming*. URL: <https://fenix.tecnico.ulisboa.pt/cursos/leic-a/disciplina-curricular/1529008373638>.
- [13] H. Geuvers. “Proof assistants: History, ideas and future.” In: *Sadhana* (2009).
- [14] C. A. R. Hoare. “An axiomatic basis for computer programming.” In: *Communications of the ACM* (1969).

## BIBLIOGRAPHY

---

- [15] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbab, 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>.
- [16] *Lógica Computacional FCT*. URL: <http://lc.ssdi.di.fct.unl.pt/1819/web/index.html>.
- [17] *Lógica Computacional UBI*. URL: <http://www.di.ubi.pt/~desousa/LC/lc.html>.
- [18] A. Ravara. "Objectives, syllabus, contents and teaching and assessment methods." In: *Report on the Curricular Unit Computational Logic* (2018).
- [19] G. J. Sussman and G. L. Steele. "Scheme: A interpreter for extended lambda calculus." In: *Higher-Order and Symbolic Computation* (1998).
- [20] *Tezos*. URL: <https://tezos.com/>.
- [21] *The Coq Proof Assistant*. URL: <https://coq.inria.fr/about-coq>.
- [22] M Thomas and H Thimbleby. "Computer Bugs in Hospitals: An Unnoticed Killer." In: (2018). URL: <http://www.harold.thimbleby.net/killer.pdf>.
- [23] *Twelf Project*. URL: [http://twelf.org/wiki/Main\\_Page](http://twelf.org/wiki/Main_Page).
- [24] *Twelf Theorem Prover*. URL: [https://www.cs.cmu.edu/~twelf/guide-1-4/twelf\\_10.html](https://www.cs.cmu.edu/~twelf/guide-1-4/twelf_10.html).
- [25] *University of Algarve*. 2019. URL: <https://www.ualg.pt/en/curso/1478>.