Animated Logic: Correct Functional Conversion to Conjunctive Normal Form[*]

António Ravara, Mário Pereira, and Pedro Barroso

NOVA LINCS and NOVA School of Sciences and Technology, Portugal

Abstract

Computational Logic is "the calculus of Computer Science" and is an essential field of this area. Courses on this subject are usually either too informal (only providing pseudo-code specifications) or overly formal (merely presenting rigorous mathematical definitions) when describing algorithms. In either case, there is an emphasis on paper-and-pencil definitions and proofs rather than on computational approaches. It is seldom the case where these courses present executable codes, even if the pedagogical advantages of using tools are well known. In this paper, we present an approach to obtain formally verified implementations of classical Computational Logic algorithms. The chosen tool for this approach is the Why3 platform since it allows implementing functions very close to their mathematical definitions, as well as it concedes a high degree of automation in the verification process. As proof of concept, we implement and prove the conversion algorithms from propositional formulae to conjunctive normal form. We apply our proposal on two variants of the algorithm: one in direct-style and another with an explicit stack structure. Being both first-order, Why3 processes the proofs straightforwardly.

1 Introduction

Motivation. Foundational courses in Computer Science, like Computational Logic, aim at presenting key basilar subjects to the education of undergraduate students. To strengthen 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.

Herein we present work developed in the context of the `FACTOR` [5] project, which aims to promote the use of OCaml [11] and correct code development practices in the Portuguese-speaking academic community. Specifically, the objectives of the project are the functional implementation of classical Computational Logic algorithms and Formal Languages, the development of correctness proofs and the step-by-step execution to help understanding the algorithm.

The algorithm for converting propositional formulae to Conjunctive Normal Form (CNF)[1] is often presented formally, with rigorous mathematical definitions that are sometimes difficult to read [4, 9, 12], or informally, intended for Computer Science but with textual definitions in non-executable pseudo-code [3, 10]. The implementation of algorithms of this nature is a fundamental piece for learning and understanding them. Languages such OCaml allow implementations very close to the mathematical definitions, helping the study because they are executable. Also, the correctness proof of a functional implementation is simpler than for the imperative one. We chose this version of the algorithm instead of a more advanced version because it is the one taught to students, thus having an important pedagogical character.

[*]Work partially supported by the Tezos Foundation through the project FACTOR, and by FCT - Foundation for Science and Technology, I.P., within NOVA LINCS, via project UID/CEC/04516/2019.

[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.

Lastly, we want second year students, more specifically in their first course of logic, to start getting used to computer-assisted proofs by seeing some correctness proofs and think they can do it. For that, the proofs need to be as automatic as possible with simple and intuitive verification conditions that do not escape the context of the course.

Contributions. As proof of concept, we implement and prove correct the referred algorithm in Why3 [6], a platform for deductive program verification. Why3 provides a first-order language with polymorphic types, pattern matching and inductive predicates, called WhyML. Also, it offers a certified OCaml code extraction mechanism and support for third-party provers.

To support the step-by-step execution of the algorithm, an important feature to help students understanding the definitions, we also implemented a version in Continuation-Passing Style (CPS) [18] and via defunctionalization got an evaluator, a version close to a first-order abstract machine [1]. Due to the limited support of Why3 to the higher order, it was not possible to close the correctness proof for the CPS version. This limitation however is not present in the defunctioned implementation that has an explicit stack structure, but in first-order. This implementation resulted from a mechanical transformation from the CPS version. This version has been naturally proven correct by Why3.

In short, this article presents pedagogical material to support the teaching of classical Computational Logic algorithms. We developed 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. Both were proved sound with small effort, basically following from "natural" assertions associated with the code to prove it correct. The repository url is https://gitlab.com/releaselab/factor/formally-verified-bug-free-implementations-of-logical-algorithms.

2 Functional presentation of the algorithm

For simplicity let us call T 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:

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

where p ranges over a set of propositional variables. The function T produces formulae in CNF, where a formula in CNF is an element of the set J_p, defined as follows:

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

Herein, we have T: $G_p \rightarrow J_p$, where T(ϕ) = cnfc (nnfc (impl_free (ϕ))).

The algorithm composes three functions: the `impl_free` function, responsible for eliminating the implications; the `nnfc` function, responsible for converting to Negation Normal Form (a formula is in NNF if the negation operator is only applied to sub-formulae that are literals); and 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:

$$\begin{aligned} H_p \triangleq\ & \psi ::= l \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \\ & l\ ::= p \mid \bot \\ I_p \triangleq\ & \epsilon\ ::= l \mid \neg l \mid \epsilon \wedge \epsilon \mid \epsilon \vee \epsilon \\ & l\ ::= p \mid \bot \end{aligned}$$

3 Supporting Boolean Theory

To support our implementation, we implemented a Boolean theory. This allows us to have clear proofs and increase the degree of automation. Since we want to have control and more in-depth knowledge of the theory, we preferred to adopt a back-to-basics strategy and build ourselves the Boolean theory.

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

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

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

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

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

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

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

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

The six properties in Table 1 define a Boolean Algebra. We refer for each the corresponding Why3 Lemma.

If we define $a + b := (a \wedge \neg b) \vee (b \wedge \neg a) = (a \vee b) \wedge \neg(a \wedge b)$ and $a\ .\ b := a \wedge b$, the Boolean Algebra induces a Boolean ring (a ring where the property $a^2 = a$ holds). The zero element of the ring coincides with the $\bot$ of the Boolean Algebra, and the multiplicative identity element with the $\top$ [19].

We also ensure the properties of the Table 2 hold the Boolean Algebra. The property that ensures that bot differs from top could also be defined, but since that would not give more information to the provers, we decide to omit it.

Property Name	**Operation**	**Boolean Property**	**Why3 Lemma**
Absorption	Conjunction	$a \wedge (a \vee b) = a$	A.1 - Page 17
	Disjunction	$a \vee (a \wedge b) = a$	A.2 - Page 17
Identity	Conjunction	$a \wedge \top = a$	A.3 - Page 17
	Disjunction	$a \vee \bot = a$	A.4 - Page 17
Associativity	Conjunction	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	A.5 - Page 17
	Disjunction	$a \vee (b \vee c) = (a \vee b) \vee c$	A.6 - Page 17
Commutativity	Conjunction	$a \wedge b = b \wedge a$	A.7 - Page 17
	Disjunction	$a \vee b = b \vee a$	A.8 - Page 17
Distributivity	Conjunction	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	A.9 - Page 17
	Disjunction	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	A.10 - Page 17
Complements	Conjunction	$a \wedge \neg a = \bot$	A.11 - Page 17
	Disjunction	$a \vee \neg a = \top$	A.12 - Page 17

Table 1: Properties of Boolean Algebra and correspondent Why3 Lemma.

Property Name		**Boolean Property**	**Why3 Lemma**
Top is equivalent to the negation of bot		$\top = \neg\bot$	A.13 - Pag.17
Double negation		$\neg\neg a = a$	A.14 - Pag.18
De Morgan's Laws	Conjunction	$\neg(a \wedge b) = (\neg a \ \vee \ \neg b)$	A.15 - Pag.18
	Disjunction	$\neg(a \vee b) = (\neg a \ \wedge \ \neg b)$	A.16 - Pag.18

Table 2: Additional properties of Boolean Algebra and correspondent Why3 Lemma.

4 Algorithm Implementation

The first step in the implementation is to define the types of the formulae according to the grammars presented in Section 2.

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

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

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

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

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

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

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

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

The set J_p has (negated) literals, disjunctions, and conjunctions at the outermost level (after a disjunction there are no conjunctions). It is represented by the type `formula_cnf`:

```
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
```

Functions. The function `impl_free` removes all implications. It is recursively defined as homomorphic in all cases, except in the implication case where it takes advantage of the propositional logic law $\mathtt{A} \rightarrow \mathtt{B} \equiv \neg\mathtt{A} \vee \mathtt{B}$. It converts the constructions of the type `formula` for those of the type `formula_wi` and does recursive calls over the arguments

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

The function `nnfc` converts formulae to NNF. It is recursively defined over a combination of constructors: applying the propositional logic law $\neg\neg\mathtt{A} \equiv \mathtt{A}$ 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.

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

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

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

The `distr` function uses the propositional logic law $\mathtt{A} \vee (\mathtt{B} \wedge \mathtt{C}) \equiv (\mathtt{A} \vee \mathtt{B}) \wedge (\mathtt{A} \vee \mathtt{C})$.

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

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

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

5 How to obtain the correctness

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

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

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

```
type valuation = i → t

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

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

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

[2]t is our Boolean type

6 Proof of correctness

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

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

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

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

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

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

```
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
  | FOr_wi phi1 phi2 | FAnd_wi phi1 phi2 → size_nonneg phi1; size_nonneg phi2
end
```

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

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

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

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

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

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

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

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

Using the Why3 `session` shell command, one obtains a table with proof times for every sub-goal. This exhaustive result is not very informative. However, adding all sub-goals times represents a theoretical worst case scenario where tasks would be proved sequentially (the real behaviour is in fact a parallel execution). Nevertheless, we will use this criterion when presenting the proof time results, as it makes easier a comparison and allow us to present more compact tables. Table 3 shows the time of aggregated proof obligation of the CNF conversion algorithm. We present the results of CVC4, Alt-Ergo and Z3, however, the last one times out trying to prove some verification conditions. For the next tables of proof times, we will only show the results for the provers that validates most of the verification condition.

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

Table 3: Aggregated proof time of CNF proof obligations.

7 Towards Step-by-Step Execution

The ability to execute in step-by-step or even to rewind or step back computations is fundamental for programmers, as it permits the inspection of the intermediate values of computations.

Moreover, forward and backwards step-wise execution provides a better understanding of the code under inspection. Previous work, also within the scope of the FACTOR project ("Rewinding functions through CPS"), shows how to support tracing functionalities in continuation-passing style programming [8]. To make use of the functionalities provide by the mentioned work, we developed versions of our implementations with explicit stack structure. Henceforth, this section presents the CPS/Defunctionalization transformation of the implementation listed in Section 4.

Continuation-Passing Style. (CPS) is a programming style where the control is passed explicitly in the form of a continuation. So, using the process to transform functions into CPS [15, 2], we have the following code for the `impl_free` function:

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

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

The process is straightforward; we add an extra parameter to the original function, in this case, `(k: formula_wi → 'a)`. Henceforth, we need to use this `k` function instead of returning the values. For the case where `phi` is `Prop t` or `Var i`, we just apply our `k` function to the corresponding constructor. When `phi` is `Neg phi1`, we write a new continuation that applies `k` to the already processed formulae (`processed_phi1`), which is then passed to the next function iterations. For the remaining cases, since there are two formulae associated with the constructor (binary constructor), we write two new continuations instead of just one. The left continuation is a direct recursion with `phi2` and the right continuation as arguments. Furthermore, the right continuation applies the `k` function to the corresponding constructor with both continuations as arguments. The complete code is in the repository. To better illustrate, Table 4 shows the execution of `impl_free_cps` for `P → Q`, step-by-step.

Step #	**Result**
Initial Call	`impl_free_cps (Impl (Var P) (Var Q)) (fun k → k)`
1st Step	`impl_free_cps (Var P) (fun impl_left →` `impl_free_cps (Var Q) (fun impl_right →` `k (FOr_wi (FNeg_wi impl_left) impl_right)))`
2nd Step	`impl_free_cps (Var Q) (fun impl_right →` `k (FOr_wi (FNeg_wi (L_wi (LVar P))) impl_right)))`
3rd Step	`k (FOr_wi (FNeg_wi (L_wi (LVar P))) (L_wi (LVar Q)))`
4th Step	`(FOr_wi (FNeg_wi (L_wi (LVar P))) (L_wi (LVar Q)))`

Table 4: Computation of `impl_free_cps` for `P → Q`.

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

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

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

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

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

The proof time for each generated proof obligation can be observed in the Table 5. Once again, if they were executed sequentially.

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

Table 5: Aggregated proof time of CNF-CPS proof obligations.

Observations. This implementation, uses types that represent grammars but, as mentioned in Section 8, undergraduates learn it only with two sets of formulae. Using these types, most correctness criterion are "automatically" ensured, because the functions output is tailored according to its properties. We only need to ensure the equivalence of the evaluation of the domain and codomain. However, with two sets of formulae we need to introduce well-formed predicates to ensure certain criteria. For example, the `wf_negations_of_literals` well-formed predicate ensures that a formula is in NNF.

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

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

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

Difficulties in completing 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 behaviour:

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

It is not possible to prove this lemma because having only access to record projections can not ensure that, in this case, the field `nnfc_formula` is the only field of this record type. Given this limitation of Why3 [7], which in this case precludes the proof of the post-condition, we have tried to compare the formula of each type with an extensional equality predicate (==) and use this predicate as post-condition instead of polymorphic structural equality (=).

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

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

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

Defunctionalization. Defunctionalization is a program transformation technique to convert high-order programs into first-order ones. Originally, it was introduced by Reynolds as a technique to transform a higher-order interpreter into a first-order one [17]. This technique has been used to derive abstract machines for different strategies of evaluation of the lambda-calculus from compositional interpreters [1]. It produces an evaluator, close to a first-order abstract machine.

The approach we follow herein is tailored for Why3 [14, 13][3]. In this work we extend the initial explorations developed by Pereira with more ambitious use-cases, validating the approach.

[3]References 13 and 14 are of the same article but in different languages: 13 is in French; 14 in Portuguese.

Transformation process. The defunctionalization technique consists of a "mechanical" transformation in two steps:

1. Get a first order representation of the function continuations, using algebraic data types, and replace the continuations with this new representation.
2. Generate a new function (`apply`) which replaces the applications of functions in the original program.

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

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

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

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

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

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

- if it is a `Left` "continuation" constructor, we call the `impl_free_defun` function with `phi2` (the right side formula of the corresponding head constructor); and the `Right` "continuation" constructor, applied to k – the continuation argument of the left constructor – and to `impl_left` – the left side already processed (without implications);
- if it is a `Right` "continuation" constructor, we recursively call the `impl_apply` function with two arguments: the application `k` and the result of applying the corresponding formula constructor to the `impl_left` and `impl_right`.

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

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

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

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

Proof of correctness. In order to specify this version, we have to stipulate a contract to the function `impl_free_defun`. The output value of this function is the result of applying the continuation `k` to the formula without implications. Henceforth, similar to the CPS version, it would be reasonable to set `result = k (impl_free phi)` as post-condition for this function.

However, since these functions can have effects, e.g., divergence, this specification could quickly introduce logical inconsistencies. Accordingly, we define an abstraction barrier between the logic and the program, restricting, therefore, the specification language. Moreover, we abstract these functions in the form of a pair of predicates that represent their precondition and post-condition [16].

So using this notation the post-condition for the function `impl_free_defun` is now:

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

This post-condition establishes a relationship between the value passed to the continuation `k` (a formula without implications) and the output (`result`). Following this methodology, we can, also, specify the anonymous functions used inside `impl_free_cps` function.

The 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` [14].

To be able to use the direct-style functions as a specification, we have created a `post` predicate that gathers the post-conditions of the direct-style function. As for the `apply` function, such predicate performs case analysis on the continuation type; and for each constructor, we copy the post-condition present in the corresponding abstraction specification [14].

For a better understanding, we subdivide our function with the related abstraction and present the specific case of the predicate. The signature of the predicate `post` is:

```
predicate impl_post (k: impl_kont) (arg result: formula_wi)
```

The base case is the identity continuation that returns the `arg` argument.

```
| KImpl_Id → let x = arg in x = result
```

The specification for the function used inside of the Neg case of our `impl_free_cps` function is:

```
| Neg phi1 → impl_free_cps phi1 (fun processed_phi1 →
        ensures { post k (KNeg_wi processed_phi1) result }
          k (FNeg_wi processed_phi1))
```

Therefore, in the predicate post, we copy the post-condition of our abstraction to the corresponding case (`KImpl_Neg`):

```
| KImpl_Neg k → let processed_phi1 = arg in
      impl_post k (FNeg_wi processed_phi1) result
```

For the remaining cases, there are two continuations, and each has its anonymous specification. Therefore, for these cases, our post predicate gathers the specific specification and applies it to the corresponding continuation constructor. Let us consider the following abstraction for the disjunction case of the `impl_free_cps` function:

```
| Or phi1 phi2 → impl_free_cps phi1 (fun impl_left →
        ensures { post k (FOr_wi impl_left (impl_free phi2)) result }
          impl_free_cps phi2 (fun impl_right →
            ensures { post k (FOr_wi impl_left impl_right) result }
              k (FOr_wi impl_left impl_right)))
```

So, applying the same methodology, each abstraction is copied to the corresponding case of the predicate post. The post-condition of the function continuation `impl_left` is now the body of the function for the case `KImpl_OrLeft`:

```
| KImpl_OrLeft k phi2 → let impl_left = arg in
      impl_post k (FOr_wi impl_left (impl_free phi2)) result
```

Lastly, the `impl_right` post-condition is applied to the `KImpl_OrRight` constructor:

```
| KImpl_OrRight k impl_left → let impl_right = arg in
      impl_post k (FOr_wi impl_left impl_right) result
```

The complete abstraction specification (anonymous specification of each continuation) and the full `post` predicate are in the project repository[4].

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

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

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

Results. The proof of correctness of the defunctionalized version of the T algorithm is naturally processed by Why3, with each proof objective being proved in less than one second as shown in Table 6.

[4]https://gitlab.com/releaselab/factor/formally-verified-bug-free-implementations-of-logical-algorithms

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

Table 6: Proof time of each defunctionalization proof obligation

8 Conclusions and Observations

Classical logical algorithms presented as recursive functions to undergraduates can have a very close functional implementation that is easy to prove correct with a high degree of automation. We develop herein an exercise: a pure functional implementation of the conversion of propositional formulae to normal conjunctive normal forms. The implementation was proved sound with small effort, basically following from the assertions one naturally associates with the code to prove it correct. We defined the theories before, with all the properties needed; therefore, the proofs were clear and naturally processed. This reinforces the conclusion that it is feasible to show to students correctness proofs of such implementations. This paper presents thus a successful proof-of-concept of formally verified bug-free implementations of (logical) algorithms.

The CPS and Defunctionalization techniques produce code with explicit stack structure since each function call return a function (continuation). These techniques give full control over program flow, allowing to, in the future, introduce a mechanism that can stop and resume the execution. The CPS transformation adds a function (continuation) as an argument, thus turning it a higher-order function. However, in Why3 it is not possible to specify the nature of the continuation arguments, and Why3 also has limitations regarding the comparison of invariant types, which hampers the verification process. The defunctionalization technique produces a close version of a first-order abstract machine. Being this version and the direct-style first-order implementations, Why3 naturally processes the proofs, as seen in Table 3 and 6 each proof obligation have been proved in less than a second.

Notwithstanding the expansion possibilities of this proof-of-concept, in the short term, we consider it is more relevant to implement the Horn algorithm and perform its correctness (ongoing) and the step-by-step execution. Lastly, it is also important to continue applying this approach to other algorithms of Computational Logic courses, like the resolution algorithm.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A Functional Correspondence Between Evaluators and Abstract Machines. In *Proceedings of PPDP'03*, pages 8–19. ACM, 2003.

[2] Andrew W Appel. *Compiling with Continuations*. Cambridge University Press, 2006.

[3] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012.

[4] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[5] FACTOR: Functional ApproaCh Teaching pOrtuguese couRses. `http://ctp.di.fct.unl.pt/FACTOR/`.

[6] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *Proceedings of Programming Languages and Systems*, pages 125–128. Springer, 2013.

[7] Add Injectivity for Type Invariant (#287) · Why3 Issues. `https://gitlab.inria.fr/why3/why3/issues/287`.

[8] Marco Giunti. Rewinding functions through cps. `https://releaselab.gitlab.io/factor/pdfs/rewind_exp_report.pdf`, 2019.

[9] Alan G. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1988.

[10] Michael Huth and Mark Dermot Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems (2. ed.)*. Cambridge University Press, 2004.

[11] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml System Release 4.07: Documentation and User's Manual. Intern report, Inria, 2018.

[12] Elliott Mendelson. *Introduction to Mathematical Logic (3. ed.)*. Chapman and Hall, 1987.

[13] Mário Pereira. Défonctionnaliser pour Prouver. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[14] Mário Pereira. Desfuncionalizar para Provar. *CoRR*, abs/1905.08368, 2019.

[15] G.D. Plotkin. Call-by-name, Call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[16] Yann Régis-Gianas and François Pottier. A Hoare Logic for Call-by-Value Functional Programs. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08, page 305–335, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of Higher-Order and Symbolic Computation*, volume 11, pages 363–397, 1998.

[18] Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[19] Marshall H Stone. *The Theory of Representation for Boolean Algebras*, volume 40. JSTOR, 1936.

A Why3 Lemmas of our Boolean Algebra representation

This lemmas defines the properties of our Boolean Algebra representation. The full code can be accessed in https://gitlab.com/releaselab/factor/formally-verified-bug-free-implementations-of-logical-algorithms/-/blob/master/booltheory.mlw.

Lemma A.1:

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

Lemma A.2:

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

Lemma A.3:

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

Lemma A.4:

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

Lemma A.5:

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

Lemma A.6:

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

Lemma A.7:

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

Lemma A.8:

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

Lemma A.9:

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

Lemma A.10:

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

Lemma A.11:

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

Lemma A.12:

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

Lemma A.13:

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

Lemma A.14:

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

Lemma A.15:

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

Lemma A.16:

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