Syntax vs Semantics: Comparing Consistency Proofs for Minimal Propositional Logics

Felipe Sasdelli
felipe.sasdelli@aluno.ufop.edu.br
Departamento de Computação
Universidade Federal de Ouro Preto

Maycon Amaro
maycon.amaro@aluno.ufop.edu.br
Departamento de Computação
Universidade Federal de Ouro Preto

Elton Cardoso
eltonmc@ufop.edu.br
Programa de Pós-Graduação em
Ciência da Computação
Universidade Federal de Ouro Preto

Samuel Feitosa
samuel.feitosa@ifsc.edu.br
Departamento de Informática
Caçador, Santa Catarina, Brazil

Rodrigo Ribeiro
rodrigo.ribeiro@ufop.edu.br
Programa de Pós-Graduação em
Ciência da Computação
Universidade Federal de Ouro Preto

ABSTRACT

Consistency is a key property of any logical system. However, proofs of consistency usually rely on heavy proof theory notions like admissibility of cut. A more semantics-based approach to consistency proofs explores the correspondence between a logic and its relationship with the evaluation in a λ-calculus, known as Curry-Howard isomorphism. In this work, we present a comparison between two formalizations of consistency for minimal propositional logic: one using a semantic-based approach and another following the (traditional) syntactic, proof-theoretical approach in both Coq proof assistant and Agda programming language. We conclude by discussing the lessons learned during the cerfication of these results in both languages.

CCS CONCEPTS

• **Theory of computation → Proof theory**; *Type theory*.

KEYWORDS

Consistency, Curry-Howard Isomorphism, Coq proof assistant

ACM Reference Format:
Felipe Sasdelli, Maycon Amaro, Elton Cardoso, Samuel Feitosa, and Rodrigo Ribeiro. 2020. Syntax vs Semantics: Comparing Consistency Proofs for Minimal Propositional Logics. In *Proceedings of XXIV BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES (SBLP2020).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

A crucial property of a logical system is consistency, which states that it does not entail a contradiction. Basically, consistency implies that not all formulas are provable. While having a simple motivation, consistency proofs rely on the well-known admissibility of cut property, which has a rather delicate inductive proof. Gentzen, in his seminal work [15], gives the first consistency proof of logic by introducing an auxiliary formalism, the sequent calculus, in which consistency is trivial. Next, Gentzen shows that the natural deduction system is equivalent to his sequent calculus extended with an additional rule: the cut rule. The final (and hardest) piece of Gentzen's proof is to show that the cut rule is redundant, i.e., it is admissible. As a consequence, we know something stronger: all propositions provable in the natural deduction system are also provable in the sequent calculus without cut. Since we know that the sequent calculus is consistent, we hence also know that the natural deduction calculus is [25].

However, proving the admissibility of cut is not easy, even for simple logics. Proofs of admissibility need nested inductions, and we need to be really careful to ensure a decreasing measure on each use of the inductive hypothesis. Such proofs have a heavy syntactic flavor since they recursively manipulate proof tree structures to eliminate cuts. A more semantics-based approach relies on interpreting logics as its underlying λ-calculus and proves consistency by using its computation machinery. In this work, we report on formalizing these two approaches for a minimal version of propositional logics.

This work results from a research project motivated by questions raised by undergraduate students on a first course on formal logics at Universidade Federal de Ouro Preto, Brazil. The students were encouraged to "find the answer" by formalizing them in proof assistant systems. After some months following basic exercises on Agda and Coq on-line text books [26, 33], they were able to start the formalization of consistency for propositional logics. This paper reports on the Coq formalization of two different approaches for consistency proofs of a minimal version of propositional logics and briefly discuss an alternative Agda formalization of the same results, also considering the conjunction and disjunction connectives. We are aware that there are more extensive formalizations of propositional logic in Coq [32] and other proof assistants [21]. However, our focus is on showing how the understanding of the Curry-Howard correspondence can lead to simple formalizations of mathematical results through its computational representation.

More specifically, we contribute:

SBLP2020, September 23–27, 2020, Natal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

- We present a semantics-based consistency proof for a minimal propositional logic in Coq. Our proof is completely represented as Coq functions using dependently-typed pattern maching in less than 90 lines of code.
- We also formalize the traditional proof theoretical cut-based proof of consistency. Unlike the semantics-based proof, this formalization required the definition of several intermediate definitions and lemmas to complete the proof. Instead of focusing on presenting tactic scripts, we outline the proof strategies used in the main lemmas to ensure the consistency.
- We formalize the same results in the context of a broader version of propositional logics in the Agda programming language and present some conclusions obtained by coding these results in a different proof assistant.

We organize this work as follows: Section 2 presents basic definitions about the minimal logic considered and Section 3 presents a brief introductions to the Coq proof assistant and the Agda programming language. Section 4 describes the semantics-based proof of consistency implemented in Coq and Section 5 presents our formalization of Gentzen's style consistency proof. We briefly discuss our Agda formalization in Section 6. Section 7 draws some lessons learned during the formalization of these consistency proofs. Finally, Section 8 presents related works and Section 9 concludes.

The complete formalization was verified using Coq version 8.10.2 and Agda formalization was checked using Agda version 2.6.1 using the standard library version 1.6.3. All these results are available on-line [3] together with the LaTeX files needed to build this paper.

2 BASIC DEFINITIONS

In this work, we consider a fragment of propositional logic which is formed by the constant *falsum* ($\bot$), logic implication ($\supset$) and variables (represented by meta-variable v), as described by the following context free grammar:

$$\alpha ::= \bot \mid v \mid \alpha \supset \alpha$$

Following common practice, we let meta-variable Γ denote contexts by a list of formulas where $\emptyset$ denotes the empty context and $\Gamma \cup \{\alpha\}$ includes the formula α in Γ. Using contexts, we can define natural deduction as an inductively defined judgment $\Gamma \vdash \alpha$ which denotes that the formula α can be deduced from the hypothesis present in Γ using the following rules:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \{Id\} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\ \{Ex\}$$

$$\frac{\Gamma \cup \{\alpha\} \vdash \beta}{\Gamma \vdash \alpha \supset \beta}\ \{\supset\text{-}I\} \qquad \frac{\Gamma \vdash \alpha \supset \beta \quad \Gamma \vdash \alpha}{\Gamma \vdash \beta}\ \{\supset\text{-}E\}$$

Rule *Id* shows that any hypothesis in Γ is provable and rule *Ex* specifies that from a contradiction we can deduce any formula. The rule $\supset$-*I* shows that we can deduce $\alpha \supset \beta$ if we are able to prove β from $\Gamma \cup \{\alpha\}$ and rule $\supset$-*E* is the well-known *modus ponens* rule.

We let notation $\Gamma \Rightarrow \alpha$ denote that α is deducible from the hypothesis in Γ using the rules of the sequent calculus which are presented next. The only difference with the natural deduction is in one rule for implication. The sequent calculus rule counter-part for implication elimination is called implication left rule, and it states that we can conclude any formula γ in a context Γ if we have that: 1) $\alpha \supset \beta \in \Gamma$; 2) $\Gamma \Rightarrow \alpha$ and 3) $\Gamma \cup \{\beta\} \Rightarrow \gamma$.

$$\frac{\alpha \in \Gamma}{\Gamma \Rightarrow \alpha}\ \{Id\} \qquad \frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow \alpha}\ \{Ex\}$$

$$\frac{\Gamma \cup \{\alpha\} \Rightarrow \beta}{\Gamma \Rightarrow \alpha \supset \beta}\ \{\supset\text{-}R\}$$

$$\frac{\alpha \supset \beta \in \Gamma \quad \Gamma \Rightarrow \alpha \quad \Gamma \cup \{\beta\} \Rightarrow \gamma}{\Gamma \Rightarrow \gamma}\ \{\supset\text{-}L\}$$

We say that the natural deduction system is consistent if there is no proof of $\emptyset \vdash \bot$. The same idea applies to sequent calculus.

3 AN OVERVIEW OF COQ AND AGDA

In this section we provide a brief introduction to the proof assistants used in the development of this work.

Coq Proof Assistant. Coq is a proof assistant based on the calculus of inductive constructions (CIC) [5], a higher order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[1], where types represent logical formulas, λ-terms represent proofs [29] and the task of checking if a piece of text is a proof of a given formula corresponds to checking if the term that represents the proof has the type corresponding to the given formula.

However, writing a proof term whose type is that of a logical formula can be a hard task, even for very simple propositions. In order to make the writing of complex proofs easier, Coq provides *tactics*, which are commands that can be used to construct proof terms in a more user friendly way.

As a tiny example, consider the task of proving the following simple formula of propositional logic:

$$(A \to B) \to (B \to C) \to A \to C$$

In Coq, such theorem can be expressed as:

```
Section EXAMPLE.
  Variables A B C : Prop.
  Theorem example : (A → B) → (B → C) → A → C.
  Proof.
      intros H H' HA. apply H'. apply H. assumption.
  Qed.
End EXAMPLE.
```

In the previous source code, we have defined a Coq section named *EXAMPLE*[2] which declares variables *A*, *B* and *C* as being propositions (i.e. with type `Prop`). Tactic `intros` introduces variables *H*, *H'* and *HA* into the (typing) context, respectively with types $A \to B$, $B \to C$ and A and leaves goal C to be proved. Tactic `apply`, used with a term *t*, generates goal *P* when there exists *t*: $P \to Q$ in the typing context and the current goal is *Q*. Thus, `apply` *H'* changes the goal from C to B and `apply` *H* changes the goal to A. Tactic `assumption` traverses the typing context to find a hypothesis that matches with the goal.

[1] Abbreviation of Brouwer, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry-Howard "isomorphism".
[2] In Coq, we can use sections to delimit the scope of local variables.

We define next a proof of the previous propositional logical formula that, in contrast to the previous proof, which was built using tactics (`intros`, `apply` and `assumption`), is coded directly as a function:

```
Definition example'
  : (A → B) → (B → C) → A → C :=
    fun (H : A → B) (H' : B → C) (HA : A) ⇒ H' (H HA).
```

As we can note, even for very simple theorems, coding a definition directly as a Coq term can be a hard task. Because of this, the use of tactics has become the standard way of proving theorems in Coq. Furthermore, the Coq proof assistant provides not only a great number of tactics but also has a domain specific language for scripted proof automation, called $\mathcal{L}$tac. More information about Coq and $\mathcal{L}$tac can be found in [5, 9].

Agda Language. Agda is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [20]. Function types and an infinite hierarchy of types, `Set 1`, where `1` is a natural number, are built-in. Everything else is a user-defined type. The type **`Set`**, also known as $\texttt{Set}_0$, is the type of all "small" types, such as **`Bool`**, **`String`** and **`List Bool`**. The type $\texttt{Set}_1$ is the type of `Set` and "others like it", such as `Set →`**`Bool`**, **`String`** `→Set`, and `Set →Set`. We have that `Set 1` is an element of the type `Set (1+1)`, for every $l \geq 0$. This stratification of types is used to keep Agda consistent as a logical theory [29].

An ordinary (non-dependent) function type is written `A →B` and a dependent one is written `(x : A) →B`, where type `B` depends on `x`, or `∀(x : A) →B`. Agda allows the definition of *implicit parameters*, i.e. parameters whose value can be infered from the context, by surrounding them in curly braces: `∀{x : A} →B`. To avoid clutter, we'll omit implicit arguments from the source code presentation. The reader can safely assume that every free variable in a type is an implicity parameter.

As an example of Agda code, let's consider the implementation of the function **example** previously given using Coq.

```
example : (A → B) → (B → C) → A → C
example f g x = g (f x)
```

Theorem proving in Agda consists of writing functions in which the type corresponds to the theorem statement. Unlike Coq, Agda has no support for tactics. However, some limited proof automation can be achieved by its Emacs-mode or using language's reflection support [19]. More information about Agda can be found elsewhere [33].

4 SEMANTICS-BASED PROOF

Our first task for formalizing the consistency is to represent formulas (type α), which are expressed by the false constant (*Falsum* constructor) and implication (*Implies* constructor). Contexts are just a list of formulas.

```
Inductive α : Set :=
| Falsum : α
| Implies : α → α → α.

Definition Γ := list α.
```

In order to represent variables, we follow a traditional approach in the programming languages community by using *De Bruijn indices* [6], a technique for handling binding by using a nameless, position-dependent naming scheme. In our natural deduction judgement we do not use a name to identify a variable, but a well-typed de Bruijn index (type *var*) which witnesses the existence of a formula α in a context Γ. This technique is well known for avoiding out-of-bound errors. Furthermore, by using well-typed representations, these kinds of errors become meta-program typing errors and are identified by the Coq's type checker. These ideas are represented by the following judgment and its Coq definition.

$$\frac{}{\alpha \in (\alpha :: \Gamma)}\ \{Here\} \qquad \frac{\alpha \in \Gamma}{\alpha \in (\beta :: \Gamma)}\ \{There\}$$

```
Inductive var : Γ → α → Type :=
| Here : ∀ G p, var (p :: G) p
| There : ∀ G p p', var G p → var (p' :: G) p.
```

The first constructor of type *var* specifies that a formula α is in the context $\alpha :: \Gamma$ and the constructor *There* specifies that if a formula α is in Γ, then we have $\alpha \in (\beta :: \Gamma)$, for any formula β.

Using the previous definitions, we can implement natural deduction rules for our minimal logic, as presented below.

```
Inductive nd : Γ → α → Type :=
| Id : ∀ G p, var G p → nd G p
| ExFalsum : ∀ G p, nd G Falsum → nd G p
| Implies_I : ∀ G p p', nd (p' :: G) p → nd G (Implies p' p)
| Implies_E : ∀ G p p', nd G (Implies p' p) → nd G p' → nd G p.
```

The first rule (*Id*) estabilishes that any formula in the context is provable and rule *ExFalsum* defines the principle *ex-falso quodlibet*, which allows us to prove any formula if we have a deduction of *Falsum*. Rule *Implies_I* specifies that from a deduction of a formula *p* from a context *p'* :: *G*, *nd* (*p'* :: *G*) *p*, we can prove the implication *Implies p' p*. The last rule, *Implies_E*, represents the well-known *modus-ponens*, which allows us to deduce a formula *p* from deductions of *Implies p' p* and *p'*.

The Curry-Howard isomorphism states that there is a correspondence between logics and functional programming by relating logical formulas to types and proofs to λ-calculus terms [29]. In order to prove consistency of a natural deduction system, we use this analogy with λ-calculus. Basically, it says that under the Curry-Howard interpretation, there is no proof for $\emptyset \vdash \bot$ (the statement of the consistency property) showing that there is no value[3] of type $\bot$. A way to ensure that a type has no value, is to reduce arbitrary terms until we have no more reductions steps to apply and that is the strategy of our semantics-based proof: build an algorithm to reduce proof terms and use it to show that there are no proofs for $\bot$.

The reduction algorithm we use is an well-typed interpreter for the simply-typed λ-calculus based on a standard model construction. The first step in the implementation is to define the denotation of a formula by recursion on its structure. The idea is to associate the empty type (*False*) with the formula *Falsum* and a function type with the formula *Implies p1 p2*, as presented next.

[3] A value is a well-typed term which cannot be further reduced according to a semantics.

```
Program Fixpoint sem_form (p : α) : Type :=
  match p with
  | Falsum ⇒ False
  | Implies p1 p2 ⇒ sem_form p1 → sem_form p2
  end.
```

Using the *sem_form* function, we can define the context semantics as tuples of formula semantics as follows:

```
Program Fixpoint sem_ctx (G : Γ) : Type :=
  match G with
  | ∅ ⇒ unit
  | (t :: G') ⇒ sem_form t × sem_ctx G'
  end.
```

Function *sem_ctx* recurses over the structure of the input context building right-nested tuple ending with the Coq *unit* type, which is a type with a unique element. Since contexts are mapped into tuples, variables must be mapped into projections on such tuples. This would allow us to retrieve the value associated with a variable in a context.

```
Program Fixpoint sem_var {G p}(v : var G p)
  : sem_ctx G → sem_form p :=
    match v with
    | Here ⇒ fun env ⇒ fst env
    | There v' ⇒ fun env ⇒ sem_var v' (snd env)
    end.
```

Function *sem_var* receives a variable (value of type *var G p*) and a semantics of a context (a value of type *sem_ctx G*) and returns the value of the formula represented by such variable. Whenever the variable is built using constructor *Here*, we just return the first component of the input context semantics, and when we have the constructor *There*, we just call *sem_var* recursively.

Our next step is to define the semantics of natural deduction proofs. The semantics of proofs is done by function *sem_nat_ded*, which maps proofs (values of type *nat_ded G p*) and context semantics (values of type *sem_ctx G*) to the value of input proof conclusion (type *sem_form p*). The first case specifies that the semantics of an identity rule proof (constructor *Id*) is just retrieving the value of the underlying variable in the context semantics by calling function *sem_var*. The second case deals with the *ExFalsum* rule: we recurse over the proof object *Hf* which will produce a Coq object of type *False*, which is empty and so we can finish the definition with an empty pattern match. Semantics of implication introduction (*Implies_I*) simply recurses on the subderivation *Hp* using an extended context (*v'* , *env*). Finally, we define the semantics of implication elimination as simply function application of the results of the recursive call on its two subderivations.

```
Program Fixpoint sem_nat_ded {G p}(H : nat_ded G p)
  : sem_ctx G → sem_form p :=
  match H with
  | Id v ⇒ fun env ⇒ sem_var v env
  | ExFalsum Hf ⇒ fun env ⇒
      match sem_nat_ded Hf env with end
  | Implies_I Hp ⇒ fun env v' ⇒ sem_nat_ded Hp (v' , env)
  | Implies_E Hp Ha ⇒ fun env ⇒
      (sem_nat_ded Hp env) (sem_nat_ded Ha env)
  end.
```

Using all those previously defined pieces, we can prove the consistency of our little natural deduction system merely by showing that it should not be the case that we have a proof of *Falsum* using the empty set of assumptions. We can prove such fact by exhibiting a term of type *nat_ded* ∅ *Falsum* → *False*[4], which is trivially done by using function *sem_nat_ded* with term *tt*, which is the value of type *unit* that denotes the semantics of the empty context.

```
Theorem consistency : nat_ded ∅ Falsum → False
  := fun p ⇒ sem_nat_ded p tt.
```

5 GENTZEN STYLE PROOF

Now, we turn our attention to formalizing the consistency proof based on the admissibility of cut in Coq. Unlike our semantics-based proof, which uses dependently typed syntax to concisely represent formulas and natural deduction proofs, we use an explicity approach for representing formulas as sequent calculus proofs. We use natural numbers to represent variables and formulas are encoded as simple inductive type which has an immediate meaning.

```
Definition var := ℕ.

Inductive α : Type :=
| Falsum : α
| Var : var → α
| Implies : α → α → α.
```

Next, we present the sequent calculus formulation for our minimal logic. The main change on how we represent the sequent calculus is in the rule for variables. We use the boolean list membership predicate *member*, from Coq's standard library, which fits better for proof automation. In order to simplify the task of writing code that uses this predicate, we defined notation $a \in G$ which means *member a G*.

The only difference with the natural deduction is in one rule for implication. The sequent calculus rule counter-part for implication elimination is called implication left rule, which states that we can conclude any formula γ in a context Γ if we have that: 1) $\alpha \supset \beta \in \Gamma$; 2) $\Gamma \Rightarrow \alpha$ and 3) $\Gamma \cup \{\beta\} \Rightarrow \gamma$. The Coq sequent calculus implementation is presented next.

```
Inductive seq_calc : Γ → α → Prop :=
| Id G a
    : (Var a) ∈ G → seq_calc G (Var a)
| Falsum_L G a
    : Falsum ∈ G → seq_calc G a
| Implies_R G a b
    : seq_calc (a :: G) b →
      seq_calc G (Implies a b)
| Implies_L G a b c
    : (Implies a b) ∈ G →
      seq_calc G a →
      seq_calc (b :: G) c →
      seq_calc G c.
```

An important property of sequent calculus derivations is the weakening, which states that it is stable under the inclusion of new hypothesis.

Lemma 1 (Weakening). If $\Gamma \subseteq \Gamma'$ and $\Gamma \Rightarrow \alpha$ then $\Gamma' \Rightarrow \alpha$.

[4]Here we use the fact that $\neg\alpha$ is equivalent to $\alpha \supset \bot$.

Proof. Induction on the derivation of $\Gamma \Rightarrow \alpha$. □

Since weakening has a straightforward inductive proof (coded as a 4 lines tactic script), we do not comment on its details. However, this proof is used in several points in the admissibility of cut property, which we generalize using the following lemma in order to get a stronger induction hypothesis.

Lemma 2 (Generalized admissibility)**.** If $\Gamma \Rightarrow \alpha$ and $\Gamma' \Rightarrow \beta$ then $\Gamma \cup (\Gamma' - \{\alpha\}) \Rightarrow \beta$.

Proof. The proof proceeds by induction on the structure of the cut formula α. The cases for when $\alpha = \bot$ and when α is a variable easily follows by induction on $\Gamma \Rightarrow \alpha$ and using weakening on the variable case. The interesting case is when $\alpha = \alpha_1 \supset \alpha_2$ in which we proceed by induction on $\Gamma \Rightarrow \alpha$. Again, most of cases are straightforward except when the last rule used to conclude $\alpha_1 \supset \alpha_2$ was $\supset$-*R*. In this situation, we proceed by induction on $\Gamma' \Rightarrow \beta$, where the only interesting cases are when the last rule was $\supset$-*L* or $\supset$-*R*. If the last rule used in deriving $\Gamma' \Rightarrow \beta$ was $\supset$-*R* we have: $\beta = a \supset b$, for some a, b. Also, we have that $\Gamma' \cup \{a\} \Rightarrow b$. By the induction hypothesis on $\Gamma' \cup \{a\} \Rightarrow b$, we have that $\Gamma \cup ((\Gamma' \cup \{a\}) - \{\alpha_1 \supset \alpha_2\}) \Rightarrow b$. Since we have $\Gamma \cup ((\Gamma' \cup \{a\}) - \{\alpha_1 \supset \alpha_2\}) \Rightarrow b$ then we also have $\Gamma \cup (\Gamma' \cup \{a\} - \{\alpha_1 \supset \alpha_2\}) \cup \{a\} \Rightarrow b$ and the conclusion follows by rule $\supset$-*R*. The case for $\supset$-*L* follows the same structure. □

Using the previously defined lemma, the admissibility of cut is an immediate corollary.

Corollary 1 (Admissibility of cut)**.** If $\Gamma \Rightarrow \alpha$ and $\Gamma \cup \{\alpha\} \Rightarrow \beta$ then $\Gamma \Rightarrow \beta$.

Proof. Immediate consequence of Lemmas 1 and 2. □

Consistency of sequent calculus trivially follows by inspection on the structure of derivations.

Theorem 1 (Consistency of sequent calculus)**.** There is no proof of $\emptyset \Rightarrow \bot$.

Proof. Immediate from the sequent calculus rules (there is no rule to introduce $\bot$). □

The next step in the mechanization of the consistency of our minimal logic is to stabilish the equivalence between the sequent calculus and the natural deduction systems. The equivalence proofs between these two formalisms are based on a routine induction on derivations using admissibility of cut. We omit its description for brevity. The complete proofs of these equivalence results can be found in our source code repository [3]. Finally, we can prove the consistency of natural deduction by combining the proofs of consistency of the sequent calculus and the equivalence between these formalisms.

Theorem 2 (Consistency for Natural Deduction)**.** There is no proof of $\emptyset \vdash \bot$.

Proof. Suppose that $\emptyset \vdash \bot$. By the equivalence between natural deduction and sequent calculus, we have $\emptyset \Rightarrow \bot$, which contradicts Theorem 1. □

6 AGDA FORMALIZATION

In this section we briefly present some details of our Agda formalization of consistency proofs for propositional logics. Since the Agda version of the consistency proof using a well-typed interpreter for the simply-typed λ-calculus is essentially the same as our Coq implementation, we will focus on the admissibility of cut version.

One important design decision of our Agda proof was how to represent contexts. While our Coq proof consider contexts as sets, i.e., operations and relations over contexts are implemented in order to not take in account the element order and their multiplicity, the Agda version implements contexts as sequences of formulas. We follow this approach mainly because it fits better as an inductive predicate for expressing permutations. Dealing with sets in Coq was easy mainly because of the facilities offered by small-scale reflection and type classes which ease evidence construction [16, 17]. The inductive type that encode context permutations is as follows:

```
data _~_ : Context → Context → Set where
  Done  : [] ~ []
  Skip  : G ~ G' → (t :: G) ~ (t :: G')
  Swap  : (t :: t' :: G) ~ (t' :: t :: G)
  Trans : G ~ G1 → G1 ~ G' → G ~ G'
```

The first rule simply states that an empty list can only be a permutation of itself. Rule `Skip` shows that if `G ~G'` then including an element in both lists allows for building a bigger permutation, rule `Swap` allows the exchange of two adjacent list elements and `Trans` guarantee that permutation is a transitive relation. Using the permutation relation, we encode an inductive type to represent the sequent calculus rules.

```
data _⇒_ : Context → Form → Set where
  init : A :: G ⇒ A
  -- some code omitted for brevity...
  exchange : G ~ D  → G ⇒ C → D ⇒ C
```

In our Agda encoding of the sequent calculus we explicitly use a constructor for using the exchange rule which specifies that permutations of contexts do not change provability. Instead of using a list provability predicate, our sequent calculus presentation demands that the formula proved by the initial sequent (constructor `init`) is the first element of the context.

In order to prove admissibility of cut using this sequent calculus encoding, we proved some lemmas, namely weakening and contraction[5]. Weakening is proved by a simple induction on the input derivation and is omitted for brevity. However, in the proof of contraction, we need to use fuel to satisfy Agda's totality check that could not identify the function as terminating.

```
⇒-contraction : Fuel → A :: A :: G ⇒ C
  → Maybe (A :: G ⇒ C)
⇒-contraction zero _ = nothing
⇒-contraction (suc n) init = just init
⇒-contraction (suc n) (AndR D D1)
    with (⇒-contraction n D)
       | (⇒-contraction n D1)
```

[5]The contraction rule allows the removal of duplicated hypothesis in context.

```
... | just x | just x1 = just (AndR x x1)
... | _      | _       = nothing
-- some code omitted for brevity
```

Essentially, the fuel parameter is just a natural number that bounds the number of recursive calls in a function definition. Using the proofs of weakening and contraction, we can implement the admissibility of cut theorem by the following Agda function.

```
⇒-cut : Fuel → G ⇒ A → A :: G ⇒ C
      → Maybe (G ⇒ C)
⇒-cut zero _ _ = nothing
⇒-cut (suc n) init E
   = ⇒-contraction n E
⇒-cut (suc n) (AndR D D1) (AndL E)
    with (⇒-cut n (⇒-weakening D) E)
...    | just x  = ⇒-cut n (AndR D D1) x
-- some code omitted for brevity
```

Instead of using nested induction (or induction on a well-founded relation [5]), we implement the admissibility of cut using fuel based recursion. While it is certainly possible to use nested inductions (like in our proof in Coq) in Agda, it would unnecessarily clutter the presentation of our results. It is worth to mention that the use of fuel in the admissibility was only necessary to please Agda's termination checker. Similar approaches were used in the context of programming languages meta-theory [2].

7 LESSONS LEARNED

The previous sections presented two different formalizations for the consistency of a minimal propositional logic in Coq and Agda proof assistants. In this section, we briefly resume the main characteristics of each approach and try to draw some conclusions on the realized proof effort.

The first proof strategy we use was inspired by the Curry-Howard correspondence and it is, in essence, a well-typed interpreter for the simply-typed λ-calculus. The consistency is ensured by constructing a term which asserts that it is impossible to build a term of type *Falsum* from an empty context, which is done by a simple call to the λ-calculus interpreter. In Coq, the complete formalization is 85 lines long and we only use the *Program* construct to ease the task of dependently-typed pattern matching, which is necessary to construct functions to manipulate richly typed structures like the type *var* or to build types from values like *sem_form* and *sem_ctx*. No standard tactic or tactic library is used to finish the formalization. The Agda version of this proof follows essentially the same idea and has around 50 lines of code.

The second strategy implements the usual proof theoretical approach to guarantee the consistency of logics. As briefly described previously, proving the admissibility of cut needs nested inductions on the structure of the cut-formula and on the structure of the sequent-calculus derivations. The main problem on proving the cut lemma is the bureaucratic adjustement of contexts by using weakening in the right places in the proof. Our proof uses some tactics libraries [9, 26] and type class based automation to automatically produce proof terms for the subset relation between contexts. Our cut-based consistency proof has around 270 lines of code without considering the tactics libraries used. The Agda version of Gentzen style consistency demanded much more effort due to the lack of proof automation which resulted in more than 700 lines to conclude the proof.

When comparing both approaches (Gentzen style vs semantic-based style), it is obvious that the first demands approximately 3 times more lines of code than the second in Coq and more than 10 times in Agda (around 700 LOC). However, while demanding more code, the cut-based proof essentially follows the ideas presented in proof-theory textbooks. One of the main difficulties when formalizing the Gentzen style proof was the correct handling of weakening. The usage of proof automation tools and Coq type classes had a great impact on the simplification of these results. The semantics-based proof rely on the relation between the minimal propositional logic and the simply-typed λ-calculus, i.e., it is necessary to understand the consequences of the Curry-Howard isomorphism.

8 RELATED WORK

Formalizations of logics. Proof assistants have been used with sucess to formalize several logical theories. van Doorn describes a formalization of some important results about propositional logic in Coq: completeness of natural deduction, equivalence between natural deduction and sequent calculus and the admissibility of cut theorem [32]. In his formalization, van Doorn considered the full syntax of propositional logic (including negation, disjuntion and conjunction) and also proved the completeness of natural deduction. In our work, we tried to keep things to a bare minimum by considering a minimalistic version of propositional logic. We intend to include the missing conectives as future work. Another formalization of propositional logic was implemented by Michaelis and Nipkow [21] which covered several proof systems (sequent calculus, natural deduction, Hilbert systems, resolution) and proved some important meta-theoretic results like: compactness, translations between proof systems, cut-elimination and model existence.

A formalization of linear logic was conducted by Allais and McBride [1]. In essence, Allais and McBride work starts from a well-scoped λ-calculus and introduce a typed representation which leads to an intuitionistic version of linear logic which uses a relation that ensure the resource control behavior of linear logic proofs. Another work which formalizes linear logic was developed by Xavier et. al. [34]. The main novelty of their work was the formalization of a focused linear logic using a binding representation called parametric high-order abstract syntax (PHOAS) [8].

Applications of proof assistants. Recently, the interest on certified interpreters was revitalized by Amin and Rompf [2], which used definitional interpreters, implemented in Coq, to prove type soundness theorems for non-trivial typed languages like System $F_{<:}$. Amin and Rompf's formalization uses fuel-based interpreters to represent semantics and they argue that presence of the artificial fuel argument does not invalidate semantics results and allow for a better distinction between timeouts, errors and normal values thus leading to stronger results.

Proof assistants have been used with some success to classical results of parsing and automata theory. A formal constructive theory of RLs (regular language) was presented by Doczkal et. al. [10].

They formalized some fundamental results about RLs. For their formalization, they used the Ssreflect extension to Coq, which features an extensive library with support for reasoning about finite structures such as finite types and finite graphs. They established all of their results in about 1400 lines of Coq, half of which are specifications. Most of their formalization deals with translations between different representations of RLs, including REs, DFAs (deterministic finite automata), minimal DFAs and NFAs (non-deterministic finite automata). They formalized all these (and others) representations and constructed computable conversions between them. Besides other interesting aspects of their work, they proved the decidability of language equivalence for all representations. Ribeiro and Du Bois [27] described the formalization of a RE (regular expression) parsing algorithm that produces a bit representation of its parse tree in the dependently typed language Agda. The algorithm computes bit-codes using Brzozowski derivatives and they proved that the produced codes are equivalent to parse trees ensuring soundness and completeness with respect to an inductive RE semantics. They included the certified algorithm in a tool developed by themselves, named verigrep, for RE-based search in the style of GNU grep. While the authors provided formal proofs, their tool show a less effective performance when compared to other approaches to RE parsing. Ridge [28] describes a formalization of a combinator parsing library in the HOL4 theorem prover. A parser generator for such combinators is described and a proof that generated parsers are sound and complete is presented. According to Ridge, preliminary results shows that parsers built using his generator are faster than those created by the Happy parser generator [18]. Firsov describes an Agda formalization of a parsing algorithm that deals with any CFG (CYK algorithm) [12]. Bernardy et al. describe a formalization of another CFG parsing algorithm in Agda [4]: Valiant's algorithm [31], which reduces CFG parsing to boolean matrix multiplication.

Type systems and type inference algorithms has been subject of several formalization efforts reported in the literature (c.f. [11, 13, 14, 23, 24, 30]). The first works on formalizing type inference are by Nazareth and Narascewski in Isabelle/HOL [23, 24]. Both works focus on formalizing the well-known algorithm W [22], but they don't provide a verified implementation of unification. They assume all the necessary unification properties to finish their certification of type inference. The work of Dubois [11] also postulates unification and proves properties of type inference for ML using the Coq proof assistant system. Nominal techniques were used by Urban [30] to certify algorithm W in Isabelle/HOL using the Nominal package. As in other works, Urban just assumes properties that the unification algorithm should have without formalizing it.

Full formalizations of type inference for ML with structural polymorphism was reported by Jacques Garrigue [13, 14]. He fully formalizes interpreters for fragments of the OCaml programming language. Since the type system of OCaml is far more elaborate than STLC, his work involves a more substantial formalization effort than the one reported in this work. Garrigue's formalization of unification avoids the construction of a well-founded relation for constraints by defining the algorithm by using a "bound" on the number of allowed recursive calls made. Also, he uses libraries for dealing with bindings using the so-called locally nameless approach [7].

9 CONCLUSION

In this work we briefly describe a Coq formalization of a semantics based consistency proof for a minimal propositional logic. The complete proof is only 85 lines long and only uses some basic dependently typed programming features of Coq. We also formalize the consistency of this simple logic in Coq using Gentzen's admissibility of cut approach which resulted in a longer formalization: the formalization has around 270 lines of code, which were much simplified by using some tactics libraries. We also report on our efforts to reproduce the same proof using the Agda programming language which, due to the lack of proof automation, demanded more lines of code to express similar results.

As future work, we intend to extend the current formalization for the full propositional logic and also other formalisms like Hilbert systems, resolution and focused versions of sequent calculus.

REFERENCES

[1] Guillaume Allais. 2018. Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *23rd International Conference on Types for Proofs and Programs (TYPES 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi (Eds.), Vol. 104. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:22. https://doi.org/10.4230/LIPIcs.TYPES.2017.1

[2] Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 14. https://doi.org/10.1145/3009837.3009866

[4] Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda. *CoRR* abs/1601.07724 (2016). http://arxiv.org/abs/1601.07724

[5] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

[6] N.G. Bruijn, de. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[7] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408. http://dblp.uni-trier.de/db/journals/jar/jar49.html#Charguéraud12

[8] Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 08)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

[9] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[10] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A Constructive Theory of Regular Languages in Coq. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 82–97.

[11] Catherine Dubois and Valérie Ménissier-Morain. 1999. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. *J. Autom. Reasoning* 23, 3-4 (1999), 319–346. https://doi.org/10.1023/A:1006285817788

[12] Denis Firsov and Tarmo Uustalu. 2014. Certified {CYK} parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming* 83, 5 - 6 (2014), 459 – 468. https://doi.org/10.1016/j.jlamp.2014.09.002 The 24th Nordic Workshop on Programming Theory (NWPT 2012).

[13] Jacques Garrigue. 2010. A Certified Implementation of ML with Structural Polymorphism. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.), Vol. 6461. Springer, 360–375. https://doi.org/10.1007/978-3-642-17164-2_25

[14] Jacques Garrigue. 2015. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science* 25, 4 (2015), 867–891. https://doi.org/10.1017/S0960129513000066

[15] Gerhard Gentzen. 1936. Die Widerspruchsfreiheit der reinen Zahlentheorie. *Math. Ann.* 112 (1936), 493–565.

[16] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *J. Formalized Reasoning* 3, 2 (2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

[17] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to make ad hoc proof automation less ad hoc. In *Proceeding of the 16th ACM*

SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 163–175. https://doi.org/10.1145/2034773.2034798

[18] Happy 2001. Happy: The parser generator for Haskell. http://www.haskell.org/happy.

[19] Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 276–301.

[20] Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*. Oxford Logic Guides, Vol. 36. Oxford Univ. Press, New York, 127–172.

[21] Julius Michaelis and Tobias Nipkow. 2017. Propositional Proof Systems. *Archive of Formal Proofs* (June 2017). http://isa-afp.org/entries/Propositional_Proof_Systems.html, Formal proof development.

[22] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

[23] Wolfgang Naraschewski and Tobias Nipkow. 1999. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23 (1999), 299–318.

[24] Dieter Nazareth and Tobias Nipkow. 1996. Formal Verification of Algorithm W: The Monomorphic Case. In *Theorem Proving in Higher Order Logics (TPHOLs'96)*, J. von Wright, J. Grundy, and J. Harrison (Eds.), Vol. 1125. 331–346. http://www4.informatik.tu-muenchen.de/~nipkow/pubs/tphol96.dvi.gz

[25] Sara Negri, Jan von Plato, and Aarne Ranta. 2001. *Structural Proof Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9780511527340

[26] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjoberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[27] Rodrigo Ribeiro and André Du Bois. 2017. Certified Bit-Coded Regular Expression Parsing. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP 2017)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3125374.3125381

[28] Tom Ridge. 2011. Simple, Functional, Sound and Complete Parsing for All Context-free Grammars. In *Proceedings of the First International Conference on Certified Programs and Proofs (CPP'11)*. Springer-Verlag, Berlin, Heidelberg, 103–118. https://doi.org/10.1007/978-3-642-25379-9_10

[3] Felipe Sasdelli, Maycon Amaro, Elton Carsodo, Samuel Feitosa, and Rodrigo Ribeiro. 2020. Proofs of Consistency for Propositional Logic Verified in Coq and Agda. Avaliable at https://github.com/lives-group/consistency.

[29] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., USA.

[30] Christian Urban and Tobias Nipkow. 2009. Nominal verification of algorithm W. In *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, G. Huet, J.-J. Levy, and G. Plotkin (Eds.). Cambridge University Press, 363–382.

[31] Leslie G. Valiant. 1975. General Context-free Recognition in Less Than Cubic Time. *J. Comput. Syst. Sci.* 10, 2 (April 1975), 308–315. https://doi.org/10.1016/S0022-0000(75)80046-8

[32] Floris van Doorn. 2015. Propositional Calculus in Coq. arXiv:math.LO/1503.08744

[33] Philip Wadler and Wen Kokke. 2019. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/.

[34] Bruno Xavier, Carlos Olarte, Giselle Reis, and Vivek Nigam. 2017. Mechanizing Focused Linear Logic in Coq. In *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017 (Electronic Notes in Theoretical Computer Science)*, Sandra Alves and Renata Wasserman (Eds.), Vol. 338. Elsevier, 219–236. https://doi.org/10.1016/j.entcs.2018.10.014