Formalizing and Computing Propositional Quantifiers

Hugo Férée
IRIF, Université Paris Cité
Paris, France

Sam van Gool
IRIF, Université Paris Cité
Paris, France

Abstract

A surprising result of Pitts (1992) says that propositional quantifiers are definable internally in intuitionistic propositional logic (IPC). The main contribution of this paper is to provide a formalization of Pitts' result in the Coq proof assistant, and thus a verified implementation of Pitts' construction. We in addition provide an OCaml program, extracted from the Coq formalization, which computes propositional formulas that realize intuitionistic versions of $\exists p\phi$ and $\forall p\phi$ from p and ϕ.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Higher order logic**; **Constructive mathematics**; **Proof theory**; **Logic and verification**.

Keywords: intuitionistic logic, propositional quantifiers, automated theorem proving, extraction, sequent calculus

ACM Reference Format:
Hugo Férée and Sam van Gool. 2023. Formalizing and Computing Propositional Quantifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3573105.3575668

1 Introduction

A central aim of logic is to understand the structure of what can be deduced from a formula, and what is required to deduce it. In particular, given a formula ϕ and an atomic proposition p appearing in it, one may ask what formulas that are independent from p entail, or are entailed by, ϕ. In second-order logic, the definition of the *propositional quantifier* $\exists p\phi$ says precisely that it is the strongest formula not containing p and entailed by ϕ, and dually, $\forall p\phi$ is the weakest formula not containing p that entails ϕ.

In classical propositional logic, these quantifiers are simply definable internally as

$$\exists p\phi \equiv \phi[\top/p] \vee \phi[\bot/p],$$
$$\forall p\phi \equiv \phi[\top/p] \wedge \phi[\bot/p].$$

However, in logics with more than two truth values, it is much less obvious how to internalize propositional quantifiers. Indeed, a surprising result of Pitts [17] says that propositional quantifiers are still definable internally in *intuitionistic* propositional logic (IPC). The proof relies on an intricate constructive definition of the propositional quantifiers, and subsequently a large case distinction to prove the correctness. While various alternative proofs, notably [10], and generalizations of Pitts' technique to other logics and proof calculi [14], have been obtained since then, the computational content of Pitts' theorem has remained under-emphasized up until now, and, to the best of our knowledge, the algorithm for computing the propositional quantifiers has never been implemented in a useable way.

Main Contribution. The main contribution of this paper is to provide a formalization of Pitts' result in the Coq proof assistant, and thus a verified implementation of Pitts' construction. We in addition provide an OCaml program, extracted from the Coq formalization, which computes propositional formulas that realize intuitionistic versions of $\exists p\phi$ and $\forall p\phi$ from p and ϕ.

Methodology. There exist two strands of proof of Pitts' theorem, one proof-theoretic [17], the other via Kripke semantics [10]. The one we follow in this paper is based on the proof-theoretic method of [17], and relies on a proof calculus for IPC known as **LJT** or **G4ip** in the literature [8, 13, 21]. The main features of the calculus **G4ip** are that it allows for a terminating proof search without loop checking, and that it does not have a contraction rule. This calculus has itself often been at the basis of the implementation of proof search for proof assistants, notably Coq's `firstorder` tactic [4], and other implementations of decision procedures for IPC. The most intricate part of Pitts' proof, and consequently also of our formalization, is the proof of correctness of the definition of propositional quantifiers, which is done by induction on the structure of a **G4ip**-proof.

The bulk of the technical work that was required for the formalization falls into several parts that we briefly indicate here, and will describe in more detail in the rest of

CPP '23, January 16–17, 2023, Boston, MA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0026-2/23/01...$15.00
https://doi.org/10.1145/3573105.3575668

the paper. First, as a necessary foundation for the later results, we formalized the definition of the sequent calculus **G4ip** and the proofs of admissibility of various inversion rules, contraction, and weakening [8]. The sequent calculus **G4ip**, and consequently Pitts' proof, make essential use of finite multisets and the fact that the Dershowitz-Manna (DM) ordering [5] on them is well-founded. For this, we rely on an existing formalization of this fact in the CoLoR Library [1]. Second, Pitts' definition of the propositional quantifiers was originally given in a declarative form [17, Table 5], involving a mutual dependent induction on the well-founded DM multiset ordering. We translate this declarative construction into a Coq definition that still clearly mirrors the pen-and-paper definition, while moreover ensuring that Coq's strict requirements about fixpoint termination are met, using Coq's `Program` library. Throughout the proof, we make use of the `std++` library of the Iris project [18] for its tactics for multiset equality resolution, as well as some custom multiset tactics written for this proof. Note that our development does not rely on any extra axiom.

Related Work. While the calculus **G4ip** has been used in proof assistants, the calculus itself and its main admissibility results have not been previously formalized, as far as we are aware. Our work on this can be placed within the context of a long-standing interest in formalizing sequent calculi and foundational results about intuitionistic logic; we highlight only a few of the existing results that are most closely related to our work here.

Herbelin and Lee [12] prove cut elimination for a first-order intuitionistic logic, using semantic methods. A syntactic proof of cut admissibility, but for classical propositional logic, was formalized in Coq by van Doorn [6]. While Coq's `firstorder` tactic [4] uses an extension of **G4ip** to solve first-order goals containing inductively defined terms, as far as we know, properties of this tactic itself have not been mechanized. We propose that the formalization of **G4ip** that we give here could serve as a stepping stone towards a future correct-by-design implementation of this tactic.

It is well-known that propositional quantifiers are related to strong forms of interpolation properties. In particular, a well-known consequence of Pitts' theorem is that intuitionistic propositional logic has the so-called *uniform interpolation property*. Deducing uniform interpolation from propositional quantifiers in addition requires the (non-uniform) Craig interpolation property. The non-uniform version of interpolation for intuitionistic logic has been formalized in Coq [2], Isabelle/HOL [19], and Nominal Isabelle [3]. Also, in very recent work by Gattinger [9] towards formalizing a proof of interpolation for propositional dynamic logic, tableaux methods for non-classical logics were formalized in Lean. Another result, which has long been known to experts in intuitionistic logic to be related to Pitts' theorem, is Ruitenburg's fixpoint theorem for IPC [20]. This result was previously formalized in Coq [15, 16], and served as one of the original sources of inspiration for the project that we report on here.

Outline. The rest of the paper is structured as follows. In Section 2 we state more precisely the main theorem (Theorem 2.1) that we formalized in this project. We then explain in Section 3 how we implemented the proof calculus **G4ip** using inductive types and multi-sets, and how we formalized the main results about this calculus. In Section 4 we describe the work done on implementing the definition of the propositional quantifiers. Section 5 outlines the main ingredients of the correctness proof of Theorem 2.1. In Section 6 we describe the extracted program for computing propositional quantifiers, and some optimizations. Some possible directions for future work that we uncovered as a result of this project are described in the final Section 7.

Source Code and Documentation. The documentation of our Coq formalization is available at

https://ipqcoq.github.io

together with an archive containing the source code. When reading this paper on screen, colored text contains a hyperlink to the corresponding Coq code.

2 Formal Theorem Statement

We will now state Pitts' theorem more precisely. Throughout this paper, by a *formula* we mean an expression built from atomic propositions $p, q, \ldots$, the constant $\bot$ and binary connectives $\wedge, \vee$ and $\rightarrow$. Given a set S of atomic propositions, we write $F(S)$ for the set of formulas whose atomic propositions are among S and we write $\vdash$ for entailment in IPC; we will discuss this notion in more detail below.

Theorem 2.1 ([17], Thm. 1)**.** *Let p be an atomic proposition and V a set of atomic propositions with $p \notin V$. For any formula $\phi \in F(V \cup \{p\})$, there exist formulas $E_p\phi$ and $A_p\phi$ in $F(V)$ such that*

1. *$\phi \vdash E_p\phi$ and for any $\psi \in F(V)$, if $\phi \vdash \psi$ then $E_p\phi \vdash \psi$,*
2. *$A_p\phi \vdash \phi$ and for any $\theta \in F(V)$, if $\theta \vdash \phi$ then $\theta \vdash A_p\phi$.*

As the notation suggests, the formula $E_p\phi$ in this theorem realizes the propositional quantifier $\exists p\phi$ and that the formula $A_p\phi$ realizes $\forall p\phi$. Note that the atomic propositions occurring in ψ and in θ are restricted to belong to a subset of those appearing in ϕ, as was also the case in Pitts' original proof. The formulas $E_p\phi$ and $A_p\phi$ also satisfy the slightly stronger property of being, respectively, *right* and *left uniform interpolants* of ϕ with respect to p; we will further remark on links with uniform interpolation in Section 7 below.

3 A Terminating Sequent Calculus

In traditional sequent calculi for IPC, such as Gentzen's system **LJ**, some care is needed to obtain a terminating proof search procedure. As pointed out in [7], a core reason for this is that upward applications of the following proof step, which combines left implication and contraction,

$$\frac{\Gamma, \phi_1 \to \phi_2 \vdash \phi_1 \qquad \Gamma, \phi_2 \vdash \psi}{\Gamma, \phi_1 \to \phi_2 \vdash \psi}$$

can lead to infinite loops. This problem was solved in [7, 21] by removing left implication and contraction from the calculus, and instead 'splitting' the above proof step into four more detailed cases, depending on the principal connective of the formula ϕ_1. Moreover, the environment Γ of a sequent is turned from a set into a multiset, in order to avoid 'hidden' applications of contraction rules: borrowing an intuition from Linear Logic, the application of a rule in this new calculus 'consumes' the formulas in the antecedent. The end result is a sequent calculus for IPC, **G4ip**, that has a terminating proof search by construction, without the need for any loop-checking.[1]

Working towards our formalization of the sequent calculus **G4ip**, we define an *environment* to be a finite multiset of formulas, where we import the multisets, and the powerful tactics for them, from the `std++` library [18]. Here, we recall that the finite multisets of a set X may be defined as finite words over X up to permutation, or alternatively as finitely-supported functions $X \to \mathbb{N}$. Following `std++`, our formalization uses the first of these two perspectives; but see Section 7 for some remarks on this.

We formalize sequent calculus proofs of **G4ip** as the inductive type `Provable` Γ ϕ, parametric in an environment Γ and a formula ϕ. Inhabitants of the type `Provable Γ ϕ` will correspond to proof trees following the **G4ip** rules ending in the sequent $\Gamma \vdash \phi$, so we introduce the notation "$\Gamma \vdash \phi$" for the type `Provable Γ ϕ`. The following definition shows our formalization in Coq, where we omit six standard rules for space reasons and to focus on the most relevant ones; for the full definition, we refer to the Coq code.

Definition 3.1 (Provability in G4ip).

```
Inductive Provable : env -> form -> Type :=
| Atom :    ∀ Γ p,
    Γ • (Var p) ⊢ (Var p)
| ExFalso : ∀ Γ ϕ,
    Γ • ⊥ ⊢ ϕ
(* six standard rules for Or, And, and Imp-right *)
| ImpLVar : ∀ Γ p ϕ ψ,
    Γ • Var p • ϕ ⊢ ψ
 -> Γ • Var p • (Var p → ϕ) ⊢ ψ
| ImpLAnd : ∀ Γ ϕ1 ϕ2 ϕ3 ψ,
    Γ • (ϕ1 → (ϕ2 → ϕ3)) ⊢ ψ
 -> Γ • ((ϕ1 ∧ ϕ2) → ϕ3) ⊢ ψ
| ImpLOr :  ∀ Γ ϕ1 ϕ2 ϕ3 ψ,
    Γ • (ϕ1 → ϕ3) • (ϕ2 → ϕ3) ⊢ ψ
 -> Γ • ((ϕ1 ∨ ϕ2) → ϕ3) ⊢ ψ
| ImpLImp : ∀ Γ ϕ1 ϕ2 ϕ3 ψ,
    Γ • (ϕ2 → ϕ3) ⊢ (ϕ1 → ϕ2) -> Γ • ϕ3 ⊢ ψ
 -> Γ • ((ϕ1 → ϕ2) → ϕ3) ⊢ ψ
where "Γ ⊢ ϕ" := (Provable Γ ϕ).
```

[1]The calculus is actually known under two names in the literature: **LJT** [7] and **G4ip** [8]. Since the name **LJT** has also been used, especially in the formal proof community, for a different terminating sequent calculus for IPC [11], we follow [8] and refer to the calculus used in this paper as **G4ip**.

We remark that a first advantage of our use of multi-sets, rather than, for example, lists, is that the formalization of provability in **G4ip** is very close to the pen-and-paper definition. For example, the rule `ImpLAnd` would be written on paper in almost exactly the same way as in Definition 3.1:

$$\frac{\Gamma, (\phi_1 \to (\phi_2 \to \phi_3)) \vdash \psi}{\Gamma, ((\phi_1 \wedge \phi_2) \to \phi_3) \vdash \psi}$$

A definition of environments as lists, on the other hand, would have to make explicit the way that a rule's principal formula occurs in the context. For example, the rule `ImpLAnd` would have to be formalized as

```
ImpLAnd : ∀ Γ1 Γ2 ϕ1 ϕ2 ϕ3 ψ,
   Γ1 ++ (ϕ1 → (ϕ2 → ϕ3)) :: Γ2 ⊢ ψ
-> Γ1 ++ ((ϕ1 ∧ ϕ2) → ϕ3) :: Γ2 ⊢ ψ
```

Explicit witnesses for Γ_1 and Γ_2 would then have to be supplied when applying this rule.

A simple way to prove termination of the calculus **G4ip**, which we will also crucially use in our formalized proof of Pitts' theorem, is to assign an integer weight to each formula, as follows.

Definition 3.2 (Weight of a formula). *The weight of a formula is inductively defined as:*

$$\begin{cases} \texttt{weight}(\bot) & = 1 \\ \texttt{weight}(q) & = 1 \\ \texttt{weight}(\phi \vee \psi) & = 3 + \texttt{weight}(\phi) + \texttt{weight}(\psi) \\ \texttt{weight}(\phi \wedge \psi) & = 2 + \texttt{weight}(\phi) + \texttt{weight}(\psi) \\ \texttt{weight}(\phi \to \psi) & = 1 + \texttt{weight}(\phi) + \texttt{weight}(\psi) \end{cases}$$

A well-founded strict preorder on the set of formulas is then obtained by putting $\phi < \psi$ iff $\texttt{weight}(\phi) < \texttt{weight}(\psi)$.

When S, T are finite multisets of a set X, we write $S \uplus T$ for their union and when $x \in X$ we use the convenient notation $S \bullet x$ for $S \uplus \{x\}$. When $<$ is a preorder on X, recall that the *Dershowitz-Manna* ordering, $\prec$, on the set of finite multisets of X may be defined as the transitive closure of the one-step relation $S \uplus T \prec S \bullet x$, where T is any finite multiset such that $t < x$ for all $t \in T$. A crucial property of this order $\prec$ is that it is well-founded whenever the original order $<$ is:

Theorem 3.3 ([5]). *If the order $<$ on X is well-founded, then $\prec$ on the finite multisets of X is well-founded.*

Theorem 3.3 has been previously mechanized in Coq as part of the multiset utilities of the CoLoR library [1], see MultisetOrder.mord_wf. Since we use the multisets defined in `std++`, which come with convenient tactics, some additional work was needed to convince Coq that `std++` multisets implement CoLoR's multiset interface to access this theorem.

In particular, we will make extensive use of the Dershowitz-Manna ordering $\prec$ on environments, derived from the well-founded order $<$ on formulas. In fact, for convenient use in our proofs, we will often use the Dershowitz-Manna ordering on *pointed* environments. By a pointed environment, we simply mean a pair (Γ, ϕ) where Γ is an environment and ϕ is a formula. Given two pointed environments (Γ, ϕ) and (Δ, ψ), we write $(\Gamma, \phi) \prec\cdot (\Delta, \psi)$ whenever the multiset $\Gamma \bullet \phi$ is $\prec$-below $\Delta \bullet \psi$. The following is now a corollary of the Dershowitz-Manna theorem that will be used crucially in our proof.

Proposition 3.4 (Well-foundedness of $\prec\cdot$)**.** *The order $\prec\cdot$ on pointed environments is well-founded.*

The well-foundedness of $\prec\cdot$ is useful because **G4ip**-rules, when read upwards, make sequents strictly descend in this order. While we did not need this fact in our Coq formalization, it is behind several of the basic tactics that we write to manipulate proofs in **G4ip**.

Proposition 3.5. *For any **G4ip**-rule with conclusion $\Gamma \vdash \phi$ and for any of its hypotheses $\Delta \vdash \psi$ we have $(\Delta, \psi) \prec\cdot (\Gamma, \phi)$.*

Indeed, any **G4ip**-rule replaces a formula in the rule's conclusion with 0, 1 or 2 formulas with smaller weight in the rule's hypothesis. One of the 'worst case' rules in this respect is, for example, `ImpLOr`, in which, reading upwards, a multiset of the form

$$\Gamma \bullet (\phi_1 \vee \phi_2) \to \phi_3$$

in the conclusion of the rule is replaced by the multiset

$$\Gamma \bullet (\phi_1 \to \phi_3) \bullet (\phi_2 \to \phi_3)$$

in the rule's hypothesis. Since the formulas $\phi_1 \to \phi_3$ and $\phi_2 \to \phi_3$ have strictly lower weight than the formula $(\phi_1 \vee \phi_2) \to \phi_3$, we have descended in the order $\prec\cdot$, and in fact we have replaced one formula in the initial multiset by just 2 formulas of smaller weight. Other rules descend in the order in a different way. For instance, `ImpLAnd` only replaces a formula with a single one, which is not structurally smaller (it has the same depth and number of connectives), but is still of lower weight.

The derivation rules of **G4ip**, as implemented in Coq by the Provable type, are compatible with multiset equivalence by construction, without the need of structural rules, i.e., a rule of the form $\Gamma \bullet \phi \vdash \psi$ may be applied also to any sequent of the form $\Gamma' \vdash \psi$ where Γ' contains ϕ. Formally, this is stated by the following typeclass declaration:

```
Instance proper_Provable :
  Proper ((≡@{env}) ==> (=) ==> (=)) Provable.
```

Here, $\equiv$ is the extensional equivalence of multisets: two multisets are equivalent if they contain the same elements with the same multiplicities. This typeclass declaration is a simple but major ingredient in all of our proofs on **G4ip** derivations as it allows to seamlessly change the antecedent of target judgment with an equivalent one. It is used every time we need to put forward a formula of the antecedent that matches the principal **G4ip**-rule or lemma that we want to apply next. The proof of `proper_Provable` crucially relies on the multiset solver tactic of the IRIS project's `std++` library [18]. We import this tactic in a slightly modified form in our proof as a custom tactic ms.

In various places, Pitts' pen-and-paper proof relies on inferences that are well-known to be valid in intuitionistic propositional logic, but are not strictly part of the definition of the proof calculus **G4ip**. In other words, [17] takes as given the equivalence between **G4ip** and other calculi for IPC, including in particular **LJ**. However, in our formalization, it was one of our design principles to keep our development self-contained and independent from 'well-known facts' about IPC. As a consequence, we had to formalize several facts about the proof calculus **G4ip** that are used implicitly in Pitts' proof. Fortunately, the essential steps were available in Dyckhoff & Negri's 2000 article [8].

Using the proof methods of [8], we formalized the *admissibility* of various rules in **G4ip**, where we recall that a proof rule R is admissible if any sequent that is provable using the rule R is still provable without it. In particular, we formalize the facts that the contraction and the usual implication left rules are admissible in the calculus **G4ip**. As a consequence, these rules may be used in arguments showing the *correctness* of the constructions below. The fact that they are not included as rules in the *definition* of the proof calculus, however, is crucial to be able to perform the dependent induction that we describe in the next section. We also note here that the admissibility of the *cut* rule [8, Sec. 6] was not needed for Pitts' proof; but we will come back to this point in Proposition 5.1 and Section 7.

4 A Mutual Dependent Induction

Pitts' construction of the propositional quantifiers is a declarative one, with E and A being defined through a mutual dependent induction, given in the paper proof by a table of rules [17, Table 5], also see Table 1 below, which gives a representative excerpt, in a notation closer to our implementation. In this section, we describe the work that was needed to turn this declarative definition first into an actual algorithm, and then into a Coq definition which has the additional constraint of requiring termination by construction.

Here, we highlight this construction not only as an implementation of Pitts' definition that was of course a necessary step for our formal proof, but also as an interesting case study in the use of the Coq proof assistant for setting up a rather complex inductive definition, which, as we will explain below (Definition 4.3) required making explicit a rather involved function signature.

We managed to achieve our definition of E and A while keeping as close as possible to the structure of the original "pen-and-paper" definition, so that anyone familiar with Pitts' article will easily see how our formalized definitions of E and A correspond to it.

***The Paper Definitions of* E *and* A.** Before explaining our formalization method, let us first briefly recall the idea behind the definitions of the formulas $E_p\phi$ and $A_p\phi$ of [17]. The formula $E_p\phi$ should represent, according to the statement of Theorem 2.1, the strongest (up to equivalence) possible p-free consequence of ϕ. It will be defined as a big conjunction over a particularly chosen *set* of p-free consequences $\mathcal{E}_p(\phi)$, which serves as a 'basis' for the p-free consequences of ϕ, in the sense that any other p-free consequence of ϕ is already a consequence of this conjunction. Elements of $\mathcal{E}_p(\phi)$ can be thought of as various 'reasons' why an arbitrary p-free formula ψ might follow from ϕ, and the definition of [17, Table 5] exhaustively enumerates all the relevant reasons. It is a priori not clear at all that this basis $\mathcal{E}_p(\phi)$ can be chosen to have finite cardinality; this is, in a sense, the main achievement of Theorem 2.1.

Essentially the same remarks apply to $A_p\phi$, except that this formula is defined as a big *disjunction* over a set $\mathcal{A}_p(\phi)$. Finally, in order to recursively define the finite sets $\mathcal{E}_p(\phi)$ and $\mathcal{A}_p(\phi)$, it turns out to be convenient for the correctness proof to generalize their arguments a bit: in Pitts' definition, the function $\mathcal{E}_p$ takes as input any environment, and the function $\mathcal{A}_p$ takes as input any pointed environment.

In summary, Pitts defines finite sets of formulas $\mathcal{E}_p(\Delta)$ and $\mathcal{A}_p(\Delta, \phi)$ by a mutual well-founded induction on the pointed environment (Δ, ϕ) in the order $\prec\cdot$, where

- $E_p(\Delta) := \bigwedge \mathcal{E}_p(\Delta)$,
- $A_p(\Delta, \phi) := \bigvee \mathcal{A}_p(\Delta, \phi)$,
- the sets $\mathcal{E}_p(\Delta)$ and $\mathcal{A}_p(\Delta, \phi)$ are defined by two tables, which encode a pattern match on the arguments.

The original table [17, Table 5] contains 9 rules for the definition of $\mathcal{E}_p(\Delta)$ and 13 rules for the definition of $\mathcal{A}_p(\Delta, \phi)$. In Table 1, we give a representative excerpt of this table, which the reader may refer to as a guiding example while we explain our implementation of the full table.

Notation. Throughout the rest of this section, and the corresponding Coq code, we fix a variable p, with respect to which the propositional quantifiers will be computed:

```
Variable p : variable.
```

***Formalizing the Definitions of* E *and* A.** We will now explain how we formalized the mathematical description above into a Coq definition. To give a quick overview, we will first describe the rules for computing $\mathcal{E}$ and $\mathcal{A}$ in Definitions 4.1 and 4.2. Definition 4.3 then provides the main definition of the formulas E and A.

Our implementation of the table's rules relies on a simple observation that these rules have a degree of *determinism* in them, as we will explain now. At first glance, it may look as though the number of E-rules that can match a given set Δ is not easily bounded; especially given rule E_5, which matches two elements of Δ simultaneously. However, it turns out that at most one rule can be applied for each element θ of Δ. In particular, rule E_5 only applies to formulas of the form $p \to \delta$, if moreover the distinguished variable p that we are eliminating also belongs to Δ.

Table 1. Excerpt of Pitts' definitions of $\mathcal{E}(\Delta)$ and $\mathcal{A}(\Delta, \phi)$, with respect to a fixed variable p.

	Δ matches:	$\mathcal{E}(\Delta)$ contains:
E_1	$\Delta' \bullet q$	$E(\Delta') \wedge q$
E_4	$\Delta' \bullet (q \to \delta)$	$q \to E(\Delta' \bullet \delta)$
E_5	$\Delta'' \bullet p \bullet (p \to \delta)$	$E(\Delta'' \bullet p \bullet \delta)$
E_6	$\Delta' \bullet (\delta_1 \wedge \delta_2) \to \delta_3$	$E(\Delta' \bullet (\delta_1 \to (\delta_2 \to \delta_3)))$
E_8	$\Delta' \bullet ((\delta_1 \to \delta_2) \to \delta_3)$	$(E(\Delta' \bullet (\delta_2 \to \delta_3)) \to A(\Delta' \bullet (\delta_2 \to \delta_3), \delta_1 \to \delta_2)) \to E(\Delta' \bullet \delta_3)$
	Δ, ϕ matches:	$\mathcal{A}(\Delta, \phi)$ contains:
A_3	$\Delta' \bullet \delta_1 \vee \delta_2, \phi$	$(E(\Delta' \bullet \delta_1) \to A(\Delta' \bullet \delta_1, \phi)) \wedge (E(\Delta' \bullet \delta_2) \to A(\Delta' \bullet \delta_2, \phi))$
A_7	$\Delta' \bullet (\delta_1 \vee \delta_2) \to \delta_3, \phi$	$A(\Delta' \bullet (\delta_1 \to \delta_3) \bullet (\delta_2 \to \delta_3), \phi)$
A_8	$\Delta' \bullet ((\delta_1 \to \delta_2) \to \delta_3), \phi$	$(E(\Delta' \bullet (\delta_2 \to \delta_3)) \to A(\Delta' \bullet (\delta_2 \to \delta_3), (\delta_1 \to \delta_2))) \wedge A(\Delta' \bullet \delta_3, \phi)$
A_{11}	$\Delta, \phi_1 \wedge \phi_2$	$A(\Delta, \phi_1) \wedge A(\Delta, \phi_2)$
A_{12}	$\Delta, \phi_1 \vee \phi_2$	$A(\Delta, \phi_1) \vee A(\Delta, \phi_2)$
A_{13}	$\Delta, \phi_1 \to \phi_2$	$E(\Delta \bullet \phi_1, \phi_2) \to A(\Delta \bullet \phi_1, \phi_2)$

The fact that at most one of the E-rules applies to any given formula $\theta \in \Delta$ will allow us to define $\mathcal{E}_p(\Delta)$ by applying a function e_rule, defined below, to each element θ of Δ. If no rule applies to a chosen θ, for example, when θ is the distinguished variable p, we will define the value of e_rule to be $\top$. This 'additional rule' is harmless, as $\mathcal{E}$ will subsequently be used in a big conjunction to define E, so that any occurrences of $\top$ will be eliminated, using that $\phi \wedge \top$ is equivalent to ϕ in IPC. On a related note, the reader may notice that our definitions below use special symbols ⊼ , ⊻ and ⇁ instead of the usual $\wedge$, $\vee$ and $\rightarrow$. These symbols are notations for functions that "optimize" these connectives, namely, the functions make_conj, make_disj, and make_impl, respectively, which already incorporate a few obvious IPC-equivalences; see Section 6 below for more details on these optimizations.

We now first give the Coq definition of the e_rule function, which formalizes the rules for the E quantifier. We would like to eventually define $E(\Delta)$ as map (e_rule EA) Δ, with the implicit guarantee that EA will only be called on inputs that are smaller than the current one.

The type of the function e_rule can thus be understood as follows: it takes as implicit argument a pointed environment (Δ, ϕ) and as first explicit argument a function EA0 which, we will assume recursively, already computes the pair $(E(\Delta'), A(\Delta', \phi'))$ for any pointed environment (Δ', ϕ') $\prec\cdot$-smaller than (Δ, ϕ).

Thus, the argument EA0 will have the following dependent type:

```
∀ pe (Hpe : pe <· (Δ, ϕ)), form * form
```

Partially applying e_rule to these input data (Δ, ϕ) and EA0, we obtain a dependently typed function:

```
@e_rule Δ ϕ EA0 : ∀ θ : form, ∀ Hin : θ ∈ Δ, form
```

Source Code 1. The type of a partially applied E-rule

In the function that remains after partial application, the argument θ is thought of as the 'focus' formula of one of the E rules, Hin is the hypothesis that θ indeed appears in the input environment, and the value computed by

```
@e_rule Δ ϕ EA0 θ Hin
```

will then be the resulting formula that $\mathcal{E}(\Delta)$ must contain according to the relevant rule of Table 1. The reason why @e_rule Δ ϕ EA0 must have this dependent type involving the additional parameter Hin, rather than just

```
∀ θ : form, form
```

will be explained at the end of this section, when we introduce our dependent version of map. Also note that, compared to Pitts' pen-and-paper definition, e_rule is supplied with an additional 'dummy' argument ϕ, which does not play a role yet directly in Definition 4.1, but will be needed to make the mutual induction in Definition 4.3 work.

Given the type of e_rule, we now list a significant part of the Coq definition of the function e_rule.

Definition 4.1 (E rule)**.**

```
Program Definition e_rule {Δ : env} {ϕ : form}
  (EA0 : ∀ pe (Hpe : pe <· (Δ, ϕ)), form * form)
  (θ: form) (Hin : θ ∈ Δ) : form
:=

  let E Δ H   := fst (EA0 (Δ, ϕ) H) in
  let A pe0 H := snd (EA0 pe0 H) in
  let Δ'      := Δ \ {[θ]} in

  match θ with
    (* E1 *)
    | Var q => if decide (p = q) then ⊤
               else E Δ' _ ⊼ q    (* E1 *)
    (* E2, E3 omitted *)
    | Var q → δ =>
        if decide (p = q) then
          if decide (Var p ∈ Δ) then
            E (Δ' • δ) _             (* E5 *)
          else
            ⊤
        else q ⇁ E (Δ' • δ) _      (* E4 *)
    (* E6 *)
    | (δ1 ∧ δ2) → δ3 => E (Δ' • (δ1 → (δ2 → δ3))) _
    (* E7 omitted *)
    (* E8 *)
    | ((δ1 → δ2) → δ3) =>
      (E (Δ' • (δ2 → δ3)) _
      ⇁ A (Δ' • (δ2 → δ3), δ1 → δ2) _)
      ⇁ E (Δ' • δ3) _
    (* a few trivial cases omitted *)
  end.
```

Note that Δ' in the Coq code corresponds to $\Delta'' \bullet p$ in rule E_5 as written in Table 1. The implementation of rule E_8, probably the most complex E-rule, shows an instance where, to calculate the value on an input of the form $\Delta = \Delta' \bullet ((\delta_1 \rightarrow \delta_2) \rightarrow \delta_3)$, the function A is called on the pointed environment $(\Delta' \bullet (\delta_2 \rightarrow \delta_3), \delta_1 \rightarrow \delta_2)$. This pointed environment is provably $\prec\cdot\ \Delta \bullet \phi$, which allows to convince Coq that the recursive call is valid.

Essentially the same implementation technique is used for the rules A_1 to A_8 and A_{10} in [17, Table 5], with A_5 is analogous to E_5, and here $\bot$ is used when no rule applies; we refer to the source code for the full implementation. However, looking at rules A_{11} to A_{13} in Table 1, one notices that they do not depend on the content of Δ, but rather on the shape of the second argument ϕ. We split off the implementation for these rules in a separate function a_rule_form, also see Definition 4.2 below. In listing the following definitions, we omit the implementations of A_1, A_2, A_4, A_5, A_6 and A_{10}.

Definition 4.2 (A rule with focus on a formula)**.**

```
Program Definition a_rule_form {Δ : env} {ϕ : form}
  (EA0 : ∀ pe (Hpe : pe ≺· (Δ, ϕ)), form * form)
  : form :=

  let E pe0 H := fst (EA0 pe0 H) in
  let A pe0 H := snd (EA0 pe0 H) in

  match ϕ with
  | Var q =>
     if decide (p = q) then ⊥
     else Var q (* A9 *)
    (* A11 *)
  | ϕ1 ∧ ϕ2 => A (Δ, ϕ1) _ ⊼ A (Δ, ϕ2) _
    (* A12 *)
  | ϕ1 ∨ ϕ2 => A (Δ, ϕ1) _ ⊻ A (Δ, ϕ2) _
    (* A13 *)
  | ϕ1 → ϕ2 => E (Δ • ϕ1, ϕ2) _ ⇀ A (Δ • ϕ1, ϕ2) _
  | Bot     => ⊥
  end.
```

Definition 4.3 (Definition of E and A)**.**

```
Program Fixpoint EA (pe : env * form)
  {wf pointed_env_order pe} :=
  let Δ := fst pe in
  (⋀ (in_map Δ (e_rule EA)),
   ⋁ (in_map Δ (a_rule_env EA)) ⊻ a_rule_form EA).
Next Obligation. apply wf_pointed_order. Defined.

Definition E pe := (EA pe).1.
Definition A pe := (EA pe).2.
```

Note that Definition 4.3 defines E and A for any pointed environment. As in [17], we can now in particular define E and A on formulas as follows:

```
Definition Ef (ψ : form) := E ({[ψ]}, ⊥)
Definition Af (ψ : form) := A ( ∅    , ψ)
```

Proving Termination. Informally speaking, the termination of the function EA of Definition 4.3 is ensured by the fact that each recursive call is obtained by the application of one of the rules, which creates only recursive calls on inputs that are ≺· than the pointed environment currently under examination. With only a constant number of recursive calls, we could have defined $\mathcal{E}$ and $\mathcal{A}$ as bouded unions and disjunctions, and Coq's `Program` scheme would have produced as many proof obligations to ensure that the calls are made on inputs that are lower in the order ≺·. However, we here require a big conjunction (resp. disjunction) over a set of formulas, so this approach did not directly work.

The pervasive underscores in Definitions 4.1 and 4.2 are placeholders for a proof that the pointed environment immediately to its left is indeed smaller than the pointed environment (Δ, ϕ) which is the function's first argument. Instead of hardcoding these proofs as terms, and in order to keep the proof readable and close to the original tabular definition, we use the `Program` scheme so that Coq generates proof obligations for each of these placeholders, and proves them using tactics. All these proof obligations are of the form:

$$\Delta \setminus \{\theta\} \bullet \theta_1 \bullet \ldots \bullet \theta_n \prec \Delta$$

where $0 \leq n \leq 2$. We solve all of them using a single tactic which first, turns such a goal into

$$\Delta \setminus \{\theta\} \uplus \{\theta_1, \ldots, \theta_n\} \prec \Delta \setminus \{\theta\} \bullet \theta$$

then applies the multiset ordering definition so that we only obtain goals of the form $weight(\theta_i) < weight(\theta)$, which are easily proven with the linear arithmetic solver **`lia`**. This reasoning is only valid under the assumption $\theta \in \Delta$, which is why we provided it to each of the rule functions.

***A Dependent Version of* `map`.** We finally explain a last ingredient of Definition 4.3, namely, the function `in_map`. Recall that, informally, the definition of the set $\mathcal{E}$ can be understood as follows:

"If the environment Δ contains a certain formula θ, then the set $\mathcal{E}$ will contain a formula `e_rule EA` θ, which is built by calling `EA` recursively on an environment of the form $(\Delta \setminus \{\theta\}) \uplus \Theta$, where each element of Θ has strictly smaller weight than θ."

However, to be able to prove to Coq that, here, it is the case that

$$\Delta \setminus \{\theta\} \uplus \Theta \prec \Delta,$$

which is what allows us to perform the required recursive call on `EA0`, requires the additional hypothesis that $\theta \in \Delta$. This is why we added an additional argument `Hin : θ ∈ Δ` to the definitions of `e_rule` and `a_rule_env` above, as we mentioned above when discussing Source Code 1.

This now leads to a different problem. In the definition of `EA` (Definition 4.3), it would be nice to be able to say that the set $\mathcal{E}(\Delta)$ is simply defined by applying `map (e_rule EA)` to Δ. However, the usual `map` function of course has simple type

```
map : (A -> B) -> list A -> list B,
```

while our definition instead requires a function of dependent type

```
∀ (Γ : list A), ∀ (f : ∀ a, (In a Γ) -> B), list B.
```

This is why we define a function that we call in_map, a dependent version of `map`. Our definition is in the special context of environments and formulas, but it is clear how this could be generalized to lists and the above dependent type. Our definition satisfies the following specification:

Proposition 4.4. *If* `A` *is a type with decidable equality, then for any environment* Γ *and any function* `f` *of dependent type* `∀ a, (In a Γ) -> A`*, if a formula* ψ *is in* `in_map Γ f` *then there is some formula* ϕ *and a proof* `Hin: ϕ ∈ Γ` *such that* `ψ = f ϕ Hin`*. Conversely, if* ϕ *is in* Γ*, then* `f ϕ Hin` *is in* `in_map Γ f` *for some proof* `Hin: ϕ ∈ Γ`.

5 The Correctness Proof

Having defined candidate formulas $E_p\phi$ (in Coq, `Ef p` ϕ) and $A_p\phi$ (in Coq, `Af p` ϕ) in the previous section, it remains to prove that they are indeed correct, i.e., that they satisfy the statement of Theorem 2.1. The correctness proof consists in proving the following three properties of the formulas $E_p\phi$ and $A_p\phi$ (cf. [17, Prop. 5]):

1. The variables in $E_p\phi$ and $A_p\phi$ are in $V(\phi) \setminus \{p\}$;
2. The sequents $\phi \vdash E_p\phi$ and $A_p\phi \vdash \phi$ are provable;
3. $E_p\phi$ is minimal and $A_p\phi$ is maximal for $\vdash$.

Our Coq development in fact proves three more general statements than the three items above, namely EA_vars, entail_correct, and pq_correct, respectively, which hold in the general context of E and A applied to pointed environments instead of formulas. This is necessary since the inductive proofs work at this level. Our final result pitts then specializes them to the formulas `Ef p` ϕ and `Af p` ϕ.

The result EA_vars is proved by induction on the well-founded order $\prec$. Thanks to our modular definition, we first prove separately that the three rule-sets build formulas that do not introduce new variables, and eliminate all occurrences of the variable p, assuming that the recursively given argument `EA0` does so, too. Then, the result follows easily by induction, as $E(\Delta)$ and $A(\Delta, \phi)$ are defined as generalised conjunctions and disjunctions of the formulas produced by these rule-sets.

For the second item, recall $E_p\phi$ is defined as $\bigwedge \mathcal{E}_p(\{\phi\})$. Thus, proving the sequent $\phi \vdash E_p(\Delta)$ amounts to proving that $\phi \vdash \psi$ for each $\psi \in \mathcal{E}_p(\Delta)$. Similarly, $A_p(\Delta, \phi)$ is defined as a disjunction, and we proving that $\Delta, A_p(\Delta, \phi) \vdash \phi$ amounts to proving $\Delta, \psi \vdash \phi$ for every $\psi \in \mathcal{A}_p(\Delta, \phi)$. This item is therefore proved once again by induction on the well-founded order, following the definition of `EA`, using a case distinction on what Δ matches in the case of E, and what Δ, ϕ matches in the case of A. The proof here strongly relies on some admissibility results from Dyckhoff and Negri [8] mentioned at the end of Section 3, like weakening, and the following proposition, which follows from admissibility of contraction, and may be viewed as special case of admissibility of cut on an implication.

Proposition 5.1 ([8], Prop 5.3)**.**

$$\forall\ \phi\ \Gamma\ \psi\ \theta,\ \Gamma \bullet (\phi \to \psi) \vdash \theta \mathrel{->} \Gamma \bullet \psi \vdash \theta.$$

Finally, the last item, pq_correct, is a special case of the following:

Proposition 5.2 (Uniformity of E and A)**.** *Let Γ be an environment not containing the variable p, let Δ be any environment and ϕ a formula such that*

$$\Gamma \uplus \Delta \vdash \phi.$$

The following two properties hold:

1. *if p does not occur in ϕ, then $\Gamma \bullet E(\Delta, \phi) \vdash \phi$,*
2. *$\Gamma \bullet E(\Delta, \phi) \vdash A(\Delta, \phi)$.*

It is the most difficult of the three, mainly as it is proved by a dependent induction on the structure of the proof of $\Gamma \uplus \Delta \vdash \phi$. There are indeed 12 rules for **G4ip**, and two derivations to build in each case, leading to about 200 lines of tactics. Most of the cases are quite simple though, and simply consist in the application of a few derivation rules, followed by the application of the induction hypotheses. The case for `ImpLVar` however is rather involved and is split into several subcases, depending on whether the principal formula is the distinguished variable p or not, and how the two formulas `Var p` and `Var p` $\to \phi$ are split between the environments Γ and Δ. Despite this abundance of cases, the proof follows exactly the structure of the original paper proof.

6 Extraction and Optimisations

Extraction. Coq's extraction mechanism provides a simple way to produce OCaml source code from a formal Coq development. In our case, this mechanism allows us to automatically compute the values of propositional quantifiers on arbitrary examples of formulas. The OCaml code we extract from our Coq proof thus contains two functions of type `form -> form` implementing `Ef` and `Af`. The source code we release with this paper contains build instructions for the extraction to an OCaml library, as well as some boilerplate OCaml code illustrating how to use the library. The library allows one to experiment with propositional quantifiers, and possibly to confirm or refute conjectures about them; we mention some first steps below.

Basic Efficiency Evaluation and Complexity Analysis. First, the computation works very efficiently for most small formulas. This was not obvious from the start, since computational complexity aspects were not considered in the original paper [17], and we did not particularly focus on having an efficient implementation in the proof stage; rather, our development was designed to be faithful to the definitions on paper, and convenient for use in formal proofs. In particular, the choice of environments as multisets from the `std++` adds many abstraction layers and induces inefficient conversions between data structures. However, our tests show that the main efficiency bottleneck is not here, but rather in the details of Pitts' algorithm itself.

Indeed, we can first notice that some formula patterns will trigger more recursive calls than others. This is in particular true for rules E_8 and A_8 which produce three recursive calls each, when the distinguished formula is of the form $(\theta_1 \to \theta_2) \to \theta_3$. It is not directly obvious to anticipate the computation time, or even the recursion depth, as it not based on a direct measure of the input formula, but on the multiset ordering. However, it is clear that the depth of implications plays a crucial role here. More precisely, a naïve implementation of Pitts' quantifiers has to make at least 3^d nested recursive calls if d is the left-most implication depth of the input formula. Performing tests on the simple

Table 2. Experimental results on computations of propositional quantifiers for the sequence of formulas ϕ_n defined in (1). The 'orig.' columns contain the original weights, and the 'opt.' columns the weight after the optimisations described in Section 6. 'SO' means stack overflow.

n	weight(ϕ_n)	weight($E_{p_0}\phi_n$)		weight($A_{p_0}\phi_n$)	
		orig.	opt.	orig.	opt.
1	3	8	3	22	5
2	5	188	28	22	5
3	7	188	28	387	62
4	9	8376	1447	387	62
5	11	8376	1447	16763	2900
6	13	SO	152137	16763	2900

sequence of formulas

$$\phi_0 := p_0, \quad \phi_{n+1} := \phi_n \to p_{n+1} \tag{1}$$

shows empirically that already for small values of n, the weight of the resulting formulas computed by our program, and thus also the computation time, blow up dramatically; see Table 2. Note that the numbers in this table only give a *upper* bound on how complex the propositionally quantified formulas need to be, by showing the weight of the formula computed by our program. We leave to future work the question of examining how these formulas might be simplified to smaller, IPC-equivalent formulas. We also note in passing that a semantic proof of Pitts' Theorem, such as the one in [10], may be simpler to formalize, but would probably compute even larger formulas and thus be less efficient for the purpose of actual computation; further see Section 7.

Optimizations. After running a few tests, it is easy to see that many produced formulas could be easily simplified. Although efficiency was not our main concern, we have implemented some easy improvements that did not harm the shape of the construction, and minimizing the amount of changes in the formal proof. For this, we have replaced syntactic conjunction, disjunction and implication symbols (see Definitions 4.1, 4.2, and 4.3) with functions $\overline{\wedge}$, $\overline{\vee}$ and $\overline{\to}$ which produce potentially smaller, but **G4ip**-equivalent formulas by taking advantage of the properties of the neutral and absorbent elements (i.e. $\bot$ and $\top$) of the $\wedge$, $\vee$ and $\to$ constructors, as well as idempotency properties. For example, for any formula ϕ, $\phi \overline{\wedge} \top = \phi$, $\phi \overline{\wedge} \bot = \bot$ and $\phi \overline{\wedge} \phi = \phi$. This is especially useful in order for rules to produce simpler formulas, but also when building $\mathcal{E}$ and $\mathcal{A}$. Indeed, the 'big conjunction' that we use to build the formula $E_p(\Delta)$ is actually defined as

$$\bigwedge l \overset{\text{def}}{=} \texttt{foldl} \ \ \overline{\wedge} \ \ \top \ \ (\text{nodup } l),$$

i.e., it removes duplicates in the list of formulas l, and folds them with $\overline{\wedge}$. In other words, we often produce equivalent, smaller formulas and above all we avoid making unnecessary costly recursive calls, thus gaining both on output shape and computation time. By proving in **G4ip** that the output of such a function is equivalent to its standard counterparts, the correctness proofs for the quantifiers still work in essentially the same way. Of course, this is only a proof of concept for the benefit of anyone interested in this aspect, and more optimisations of this shape could be added. In this direction, a more structural approach would be to prove a statement of the form: if E_1 is a correct definition of the formula $\exists_p\phi$, and E_2 is provably equivalent to E_1, then E_2 is also a correct definition of $\exists_p\phi$; and similarly for $\forall_p$. These statements are easy to prove on paper but we did not yet formalize them.

We briefly note some further potential performance optimizations for the extracted OCaml program that we have not yet implemented. First, it may be beneficial to use memoization of previously calculated results, especially after normalization of the inputs. A final simple, but somewhat trivial, optimization would be to first check if p appears at all in the formula ϕ. If it does not, then ϕ itself already realizes the properties we require of both $E_p(\phi)$ and $A_p(\phi)$. This test is linear in the size of ϕ and can avoid a potentially exponential computation.

7 Conclusions and Future Work

In this paper, we reported on our mechanization in Coq of a process for encoding propositional quantifiers for formulas of intuitionistic propositional logic within the syntax of IPC itself, with a formally verified correctness proof, based on the paper proof [17]. In the process, we also formalized the notion of provability in the sequent calculus **G4ip** and the proofs from [8] of the admissibility of various rules in this calculus, including contraction and a restricted form of cut. Moreover, the extracted OCaml program provides an implementation of the computation of propositional quantifiers, making it possible for the first time to experimentally investigate open questions regarding propositionally quantified formulas and their complexity, since by-hand computations quickly turn infeasible.

An obvious first direction for future work is to also implement the admissibility of the cut rule, and the explicit algorithm for eliminating cuts in the calculus **G4ip** [8, Sec. 6]. We did not need cut-admissibility for the main result of this paper, but having a formalization of it would be a main step towards connecting it explicitly to other proof calculi for IPC, notably **LJ**. This would also enable us to formalize some further corollaries of Pitts' theorem, notably the *uniform interpolation theorem.* This theorem states that, for any $\phi \in F(V \cup \{p\})$ and for *any* formulas ψ, θ not containing p but *possibly containing other atomic propositions that are not in* V, if $\phi \vdash \psi$ then $\exists_p\phi$ is a formula such that $\phi \vdash \exists_p\phi \vdash \psi$ and if $\theta \vdash \phi$ then $\forall_p\phi$ is a formula such that $\theta \vdash \forall_p\phi \vdash \phi$. This result in addition requires Craig's interpolation property for intuitionistic logic, which has previously been formalized in Coq using the calculus **LJ** [2], but not using **G4ip**.

While we regard the formalization in Coq of Theorem 2.1 and the extracted program computing propositional quantifiers as important milestones, some improvements to our formalization are clearly still possible. In the current version, we did not yet fully separate the abstraction layer of multisets from its syntactic implementation. For example, in our current development, it is not automatically recognized that, when Γ is a multiset and $\phi \in \Gamma$ multisets Γ, then Γ is the same multiset as $(\Gamma \setminus \{\phi\}) \bullet \phi$. This type of phenomenon made it necessary to develop various custom tactics that are just permuting the elements of a multiset, in order to bring the appropriate elements "forward" and "backward" for use in a sequent proof. We conjecture that in a future version these tactics could be replaced by defining an appropriate `Setoid` and thus making "obvious" equalities of multisets transparent in the formalization.

An additional possible improvement, which occurred to us only when we were already in the final stages of the formalization work, is that the use of the Dershowitz-Manna theorem (Theorem 3.3 above) might be avoided by introducing a more refind notion of weight on multisets. This improvement relies on an observation on the calculus **G4ip** and its use in Pitts' proof, namely, that any multiset that occurs in the hypothesis of a rule is not only $\prec\cdot$ than the multiset in the conclusion, but that the size of the difference between the two multisets can in fact explicitly bounded (by 2). We leave it to future work to implement this improvement and to examine whether it also enhances the performance and complexity of the extracted OCaml program. As a methodological point, we believe this improvement will provide an interesting and typical example of finding an improvement to a theoretical, pen-and-paper procedure through the activity of formalizing it in a proof assistant.

As it stands, our construction of Pitts' quantifiers here is clearly very sensitive to the precise proof calculus that is used, and as such, it looks like a fairly special result, although a fundamental one. In recent work, Iemhoff [14] generalizes Pitts' result on the existence of propositional quantifiers to proof calculi for substructural logics, and obtains a result of the form: if a so-called "centered" proof calculus exists for a logic L, then L admits the encoding of propositional quantifiers in its syntax. Iemhoff applies this result to show the impossibility of designing sequent calculi for various intuitionistic modal logics. It would be interesting to similarly generalize our Coq formalization of Pitts' theorem to give an encoding of propositional quantifiers parametric in an arbitrary centered calculus, thus mechanizing and formalizing Iemhoff's result.

It is well-known that proofs in the Gentzen calculus **LJ** correspond to terms in simply typed λ-calculus through the Curry-Howard isomorphism. Other sequent calculi for intuitionistic logic have also been shown to correspond to λ-terms [11], and although we are not aware of an explicit such encoding for **G4ip**, our formalization in Coq of the calculus shows that such an encoding is possible in the calculus of inductive constructions. This encoding gives, for any intuitionistic formula A, a set of λ-terms of type A that are encodings of **G4ip**-proofs of A. Pitts' theorem in particular yields a construction that maps any such **G4ip**-term to a **G4ip**-term of type $\exists pA$. A curious question to be investigated further, then, is what the correct type-theoretic interpretation of Pitts' theorem should be. In particular, is there a sense in which the Pitts' propositional quantifiers give a proof-irrelevant version of the quantification over types of Girard's system **F**? And if so, what is the proof-relevant version of Pitts' theorem?[2]

Finally, Pitts' theorem for IPC is closely related to the existence of adjoints for homomorphisms between finitely presented Heyting algebras, as already noticed in [17] and much further explored in [10]. This algebraic interpretation of the statement of Pitts' theorem leads, in [10], to a semantic proof of the existence of propositional quantifiers, which involves an intricate construction on Kripke frames. The bounds on the complexity of the propositional quantifier obtained by this alternative proof method are worse than those given by the proof-theoretic method followed in this paper. However, it remains to be seen how easy or difficult it would be to formalize the semantic-style proof in a proof assistant.

References

[1] Frédéric Blanqui, Solange Coupet-Grimal, William Delobel, Sébastien Hinderer, and Adam Koprowski. 2006. CoLoR: a Coq Library on Rewriting and termination. In *Eighth International Workshop on Termination-WST 2006.*

[2] Sylvain Boulmé. 1996. *A proof of Craig's Interpolation Theorem in Coq.* Technical Report. Edinburgh University.

[3] Peter Chapman, James McKinna, and Christian Urban. 2008. Mechanising a Proof of Craig's Interpolation Theorem for Intuitionistic Logic in Nominal Isabelle. In *AISC Conference.* 38–52.

[4] Pierre Corbineau. 2003. First-Order Reasoning in the Calculus of Inductive Constructions. In *TYPES Conference (Lecture Notes in Computer Science, Vol. 3085)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer, 162–177. https://doi.org/10.1007/978-3-540-24849-1_11

[5] Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Commun. ACM* 22, 8 (aug 1979), 465–476. https://doi.org/10.1145/359138.359142

[6] Floris van Doorn. 2015. Propositional Calculus in Coq. (2015). arXiv:1503.08744

[7] Roy Dyckhoff. 1992. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic* 57, 3 (1992), 795–807.

[8] Roy Dyckhoff and Sara Negri. 2000. Admissibility of Structural Rules for Contraction-Free Systems of Intuitionistic Logic. *The Journal of Symbolic Logic* 65, 4 (2000).

[2] We thank A. Guatto for pointing us to this question.

[9] Malvin Gattinger. 2022. A Verified Proof of Craig Interpolation for Basic Modal Logic via Tableaux in Lean. In *Advances in Modal Logic*.

[10] S. Ghilardi and M. Zawadowski. 2002. *Sheaves, Games, and Model Completions*. Springer, Dordrecht.

[11] Hugo Herbelin. 1995. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic*. Springer Berlin Heidelberg, 61–75.

[12] Hugo Herbelin and Gyesik Lee. 2009. Forcing-Based Cut-Elimination for Gentzen-Style Intuitionistic Sequent Calculus. In *WoLLIC conference (Lecture Notes in Computer Science, Vol. 5514)*. Springer, 209–217. https://doi.org/10.1007/978-3-642-02261-6_17

[13] J. Hudelmaier. 1988. *A Prolog program for intuitionistic logic*. Technical Report. University of Tübingen. SNS-Bericht 88-28.

[14] Rosalie Iemhoff. 2019. Uniform interpolation and the existence of sequent calculi. *Annals of Pure and Applied Logic* 170, 11 (2019), 102711. https://doi.org/10.1016/j.apal.2019.05.008

[15] Tadeusz Litak. 2015. Formalisation of Wim Ruitenburg's paper "On the Period of Sequences (An(p)) in Intuitionistic Propositional Calculus" (JSL 1984). (2015). https://git8.cs.fau.de/software/ruitenburg1984.

[16] Tadeusz Litak. 2017. The periodic sequence property. In *TACL Conference*. https://www.cs.cas.cz/tacl2017/abstracts/TACL_2017_paper_120.pdf

[17] Andrew M. Pitts. 1992. On an Interpretation of Second Order Quantification in First Order Intuitionistic Propositional Logic. *J. Symb. Log.* 57, 1 (1992), 33–52.

[18] Iris Project. 2022. The Iris 4.0 Reference. https://plv.mpi-sws.org/iris/appendix-4.0.pdf.

[19] Tom Ridge. 2006. Craig's Interpolation Theorem formalised and mechanised in Isabelle/HOL. (2006). arXiv:cs/0607058.

[20] W. Ruitenburg. 1984. On the period of sequences (an(p)) in intuitionistic propositional calculus. *Journal of Symbolic Logic* 49, 3 (1984), 892–899.

[21] N. N. Vorobev. 1970. A new algorithm for derivability in the constructive propositional calculus. *Amer. Math. Soc. Transl.* 94 (1970), 37–71. Translated from the 1952 Russian original.

Received 2022-09-21; accepted 2022-11-21