Noname manuscript No.
(will be inserted by the editor)

Verified Model Checking for Conjunctive Positive Logic

Alex Abuin · Unai Diaz de Cerio · Montserrat Hermo · Rustan Leino · Paqui Lucio

the date of receipt and acceptance should be inserted later

Abstract We formalize, in the Dafny language and verifier, a proof system PS for deciding the model checking problem of the fragment of first-order logic, denoted $\mathcal{FO}(\forall, \exists, \wedge)$, known as Conjunctive Positive Logic (CPL). We mechanize the proofs of soundness and completeness of PS ensuring its correctness. Our formalization is representative of how various popular verification systems can be used to verify the correctness of rule-based formal systems on the basis of the least fixpoint semantics. Further, exploiting Dafny automatic code generation, from completenes proof we achieve a mechanically verified prototype implementation of a proof search mechanism that is a model checker for CPL. The model checking problem of $\mathcal{FO}(\forall, \exists, \wedge)$ is equivalent to the quantified constraint satisfaction problem (QCSP), and it is PSPACE-complete. The formalized proof system provides a way of detecting tractability cases for the general QCSP and it can be applied to arbitrary formulae of CPL.

Keywords Conjunctive Positive Logic · Quantified Constraint Satisfaction Problem · Proof System · Model Checking · Verification · Dafny.

Corresponding Author: Paqui Lucio

Paqui Lucio
Computer Languages and Systems, University of the Basque Country, San Sebastián, Spain
E-mail: paqui.lucio@ehu.eus

A. Abuin · U. Diaz de Cerio
Dependable Embedded Systems, Ikerlan Research Center, Mondragón, Spain E-mail: {aabuin,UDiazCerio}@ikerlan.es

Montserrat Hermo
Computer Languages and Systems, University of the Basque Country, San Sebastián, Spain
E-mail: montserrat.hermo@ehu.eus

Rustan Leino
Amazon Web Services, Seattle, USA E-mail: leino@amazon.com

1 Introduction

Model checking [14, 40] is the problem of deciding whether a logical sentence holds for a structure or not. It is a fundamental computational task that appears in areas such as computational logic, verification, artificial intelligence, constraint satisfaction, and computational complexity. The case where the logical sentence is a first-order sentence and the structure is finite is of interest mainly in database theory. In general, model checking problem is intractable. To be precise, model checking for first-order logic is PSPACE-complete [46]. All in all, model checking for fragments of first-order logic appears as an important challenge.

The (quantified) conjunctive positive fragment of first-order logic, in symbols $\mathcal{FO}(\forall, \exists, \wedge)$, contains all first-order sentences built on atoms using only logical symbols in $\{\forall, \exists, \wedge\}$, where an atom is the application of a predicate $R(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variable symbols (in a fixed countable set) and R is a relation (or predicate) symbol. This fragment is commonly called *Conjunctive Positive Logic (CPL)*. The fragment $\mathcal{FO}(\exists, \wedge)$ is called *Existential Conjunctive Positive Logic* and its model checking problem is equivalent to the much-studied *Constraint Satisfaction Problem (CSP)*, whereas the model checking problem of $\mathcal{FO}(\forall, \exists, \wedge)$ is equivalent to the *Quantified Constraint Satisfaction Problem (QCSP)*[35].

CSP provides a general framework in which a wide variety of combinatorial search problems can be expressed in a natural way [17, 18]. An instance of the CSP can be viewed as a collection of predicates over a set of variables. The aim is to determine whether there exist values for all of the variables such that all of the specified predicates hold simultaneously. Henceforth, from a logic approach, the CSP is viewed as the model checking problem for $\mathcal{FO}(\exists, \wedge)$. This approach has proven to be very successful due to the connection between the logic notion of definability and the complexity of the CSP [11]. The CSP is NP-hard (actually, it is NP-complete). Indeed, in [12] a 3SAT instance is expressed by a CSP instance where all variables range over a boolean domain and predicates correspond to the clauses (thus the arity of each predicate is 3). Although the CSP is NP-complete in general, there are additional restrictions on the input instances that make the problem easier. One of the main aims of research in CSP is to identify and classify tractable special cases of the general problem. The theoretical literature on CSP mainly investigates two kind of restrictions. The first type is to restrict the type of predicates that are allowed. This direction includes the classical work of Schaefer [43] and its many generalizations. The second type is to restrict the structure induced by the predicates on the variables [25, 38]. QCSP is a natural generalization of the CSP and it can be viewed as the model checking problem for Conjunctive Positive Logic (or $\mathcal{FO}(\forall, \exists, \wedge)$). A study of complexity of the model checking problem of various fragments of first-order logic can be found in [36], whereas a good, and quite recent, survey on the QCSP and close problems is [37]. QCSP is actively studied in artificial intelligence, where it is used to model problems, for example, in non-monotonic reasoning [19] and in planning [42]. Various general (superpolynomial or incomplete) algorithms for the QCSP over the boolean domain have been suggested [9, 10, 24, 50], and quite recently researchers have started investigations on solving non-boolean QCSP problems [7, 23, 34, 50]. Since QBF can be expressed as a QCSP instance [12], the general QCSP is PSPACE-complete. Like in the CSP case, a lot of research is being done nowadays trying to find tractable instances.It is in this context where a proof system for QCSP, called PS, was introduced in [1].

This proof system provides a way of detecting tractability cases for the general QCSP. PS is a slight variant of the proof system previously defined in [13]. The study of the proofs that can be generated by PS is a good tool to discover lower bounds in proof complexity, and even on the running time of algorithms that determine the satisfiability of formulas. A good understanding of how PS proofs are generated provides clues on the very nature of PSPACE-complete problems [13]. So far, the main motivation to formalize PS. Besides, the restriction of PS to the boolean domain turns out PS into a proof system that simulates Q-resolution [13], which many of the so-called QBF-solvers are based on. Q-resolution was introduced in [9]. After, many different extensions and variants has been proposed, such as Long-Distance resolution [3, 52], QU-resolution [22], and LQU-resolution [4] which combines Long-Distance and universal resolution. It is worth noting that all these systems are defined in the propositional setting whereas PS works over any finite domain. This is a strength of PS because some scenarios are more naturally and cleanly modelled by allowing variables to be quantified over domains of size greater than two.

Automated reasoners have turned out to be useful tools in a wide range of areas from pure mathematics to smart contracts. Automated reasoners play a vital role in formalizing and certifying computation related engines, such as compilers, virtual machines, operating systems, protocols, programming languages, solvers, checkers, etc. Very often, formalizations are very long and complicated, and certificate proofs are error-prone and difficult to check by hand. Henceforth, there is genuine value in having mechanized (machine-checked) proofs. *Interactive theorem provers* or *proof assistants*, such as Agda [8], Coq [49] and Isabelle/HOL [39], have been successfully used for this task for many years, producing an extensive collection of system formalizations and mechanized proofs. An excellent extensive review on proof-assistants developments of different kinds of software systems is [41]. Machine-checked formalizations of logical systems [51] and checkers (or solvers) [5, 20, 44, 45] are quite recent. In [20, 45] executable code is generated from the verified formalization of the system. *Automatic program verifiers* –such as ACL2 [27], VCC [16], F* [47], VeriFast [26], Why [21] and Dafny [28]– are dedicated reasoners to verify behavioral properties of programs written in some specific programming language. It has been recently shown [15] that program verifiers environments are also suitable for formalization of rule-based systems. Consequently, the program verification 'style' has joined the challenge of formalizing logical systems and automatically generating the code of verified checkers or solvers. Proving meta-properties of proof systems –such as soundness, completeness, and many others related to proof search– makes heavy use of advanced logic constructs, thus typically involves complex reasoning steps, beyond first-order logic. Program verifiers has extended their specification language (beyond first-order logic) with, among other, constructions that allow to reason about fixpoints in an automated way. Fixpoint reasoning is crucial to encode rule-based systems (hence, logical systems) and to prove meta-logical properties of the inference system, respectively. The reason for that is that well-founded (or terminating) recursive functions and predicates (i.e. whose recursive calls are made on arguments that are structurally smaller) are, in general, not expressive enough to represent the set of all the statements that can be proved using a set of rules. In other words, the least derivability relation induced by the a set of inference rules cannot be defined using well-founded recursion. Proof assistants provide, since long, support for fixpoint reasoning, typically with user-interaction. More recently a mostly-automatic kind of fixpoint reasoning has been introduced in program verification tools. In [6] the authors explain some examples using fixpoint formalizations in Why3. In [15] the first formalization of a rule-based system, using mostly-automatic fixpoint reasoning, is introduced. In [33] fixpoint reasoning for Dafny was introduced, providing a novel support for automatically proving lemmas using fixpoint induction. Consequently, Dafny provides a strong support to formalize logical systems, to verify its soundness and completeness (and other interesting properties), and also to generate code for their corresponding provers, checker or solvers. In addition, a significant challenge to construct large mechanized proofs is the ability to control the logical context of the proved properties, in two senses. On one hand, for clarity and easy human reading, well-defined dependencies between definitions and properties are really helpful. On the other hand, the performance of automated provers is improved as the set of logical premises needed to prove a lemma is well delimited. Dafny also provides a module system that allows the user to split formalizations into small components and to make explicit scopes and dependencies. Another Dafny feature we exploit in this work is automatic code generation that allows to generate .NET code for any verified program. To the best of our knowledge, there is no published work that substantiates all these Dafny features by presenting a (modular) formalization of a dedicated formal system and the prover-style tool obtained by automatic code generation.

In this paper we present a Dafny formalization of the proof system [1], called PS, the machine-checked proofs of its soundness and completeness, and the model checker obtained by automatic code generation. Along the presentation, we expose the constructors used inside Dafny to encode the system and to prove the main lemmas. We emphasize the fixpoint reasoning from, both, the theoretical view applied to PS and its practical use in proving the soundness of PS. We also report on our experience doing this work. The MVS-project can be downloaded from site `http://github.com/alexlesaka/VMC_CPL`, and the verified model checker is available as a web application at `http://qcspmc.ikerlan.es`.

Outline of the paper. In Section 2 we introduce the proof system PS and its least fixpoint operator. In Section 3 we provide basic notions of the Dafny language and verifier. In Section 4 we describe the formalization of the proof system PS as an inductive predicate with all the technical details. In Section 5, we explain the main ideas behind the mechanized proofs of soundness and completeness. In Section 6 we explain the structure of modules and its dependencies of our formalization, whereas in Sections 7 and 8 we respectively give implementation and experience details.

2 A Proof System for QCSP

In this section we introduce the proof system PS, along with all the necessary basic notions on QCSP, taken from [1,13]. We also relate PS with the least fixpoint of the derivability relation.

We focus on the sublogic of relational first-order logic known as Conjunctive Positive Logic (CPL). A *signature* σ is a finite set of relation symbols; each relation symbol $R \in \sigma$ has an associated arity $\mathsf{ar}(R)$ which is an element of $\mathbb{N}$. An atom is an application of a predicate $R(x_1 \dots x_{\mathsf{ar}(R)})$, where $x_1 \dots x_{\mathsf{ar}(R)}$ are variable symbols (in a fixed countable set) or constant symbols, and $R \in \sigma$. A formula (over signature σ) is built from atoms (over σ), conjunction ($\wedge$), universal quantification ($\forall$), and existential quantification ($\exists$). A *sentence* is a formula having no free variables.

A *structure* $\mathbf{B}$ on signature σ consists of a *domain* B of $\mathbf{B}$, which is a finite non-empty set and, for each symbol $R \in \sigma$, a relation $R^{\mathbf{B}} \subseteq B^{\mathsf{ar}(R)}$. We call an *interpretation* to the map that associates to each symbol R a relation $B^{\mathsf{ar}(R)}$. For a structure $\mathbf{B}$ and a sentence ϕ over the same signature, we write $\mathbf{B} \models \phi$ if the sentence ϕ is true in the structure $\mathbf{B}$.

A QCSP instance is a pair $(\phi, \mathbf{B})$ where ϕ is a sentence in CPL, and $\mathbf{B}$ is a structure, such that all the relation symbols in ϕ belong to the signature of $\mathbf{B}$. The QCSP is the problem of deciding, given a QCSP instance $(\phi, \mathbf{B})$, whether or not $\mathbf{B} \models \phi$.

Example 1 We show that 3-QBF –the case of the QBF problem where every clause has exactly three literals– can be expressed as a QCSP. Define the relations $R_{0,3}, R_{1,3}, R_{2,3}$, and $R_{0,3}$ by

$$\begin{aligned} R_{0,3} &= \{0,1\}^3 \setminus \{(0,0,0)\}, \\ R_{1,3} &= \{0,1\}^3 \setminus \{(1,0,0)\}, \\ R_{2,3} &= \{0,1\}^3 \setminus \{(1,1,0)\}, \\ R_{3,3} &= \{0,1\}^3 \setminus \{(1,1,1)\}, \end{aligned}$$

Then, for any variables x, y, z, we have the following equivalences:

$$\begin{aligned} R_{0,3}(x,y,z) &= (x \vee y \vee z), \\ R_{1,3}(x,y,z) &= (\neg x \vee y \vee z), \\ R_{2,3}(x,y,z) &= (\neg x \vee \neg y \vee z), \\ R_{3,3}(x,y,z) &= (\neg x \vee \neg y \vee \neg z), \end{aligned}$$

in the sense that, for example, the constraint $R_{1,3}(x,y,z)$ is satisfied by an assignment if and only if the clause $(\neg x \vee y \vee z)$ is satisfied by the assignment. In general, let σ be the signature $\{R_{0,3}, R_{1,3}, R_{2,3}, R_{3,3}\}$ and $\mathbf{B}$ the structure with domain $=\{0,1\}$ and such that $R^{\mathbf{B}}_{0,3}, R^{\mathbf{B}}_{1,3}, R^{\mathbf{B}}_{2,3}$, and $R^{\mathbf{B}}_{3,3}$ are defined as above. Every instance of the 3-QBF problem can be readily translated into an instance of QCSP having the same satisfying assignments. For example, the 3-QBF instance

$$\forall s\, \exists t\, \forall u\, \exists v\, ((\neg u \vee s \vee \neg t) \wedge (\neg s \vee t \vee v) \wedge (s \vee t \vee \neg v) \wedge (v \vee u \vee s)).$$

is equivalent to the QCSP instance $(\varphi, \mathbf{B})$ where

$$\varphi = \forall s\, \exists t\, \forall u\, \exists v (R_{2,3}(u,t,s) \wedge R_{1,3}(s,t,v) \wedge R_{1,3}(v,s,t) \wedge R_{3,3}(v,u,s)).$$

For our purposes, formulas are seen as trees. The proof system enables to derive what we call *constraints* at the various nodes of the tree. To facilitate the discussion, we will assume that each sentence ϕ has, associated with it, a set I_ϕ of *indices* that contains one index for each subformula occurrence of ϕ, that is, for each node of the tree corresponding for ϕ. In other words, we use an indexing, by a set I_ϕ, of the tree that represents a formula ϕ. Let us remark that (in general) the collection of constraints derivable at an occurrence of a subformula does not depend only on the subformula and on the structure, but also on the subformula's

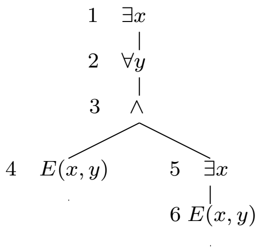

Fig. 1 Formula discussed in Example 2 (from [13]).

location in the full formula ϕ. When i is an index, we use $\phi(i)$ to denote the actual subformula of the formula occurrence corresponding to i; we will also refer to i as a *location.*

Example 2 Consider the sentence $\phi = \exists x \forall y (E(x, y) \wedge (\exists x E(x, y)))$ (see Figure 1). When viewed as a tree, this formula has 6 nodes. We can index the representation of ϕ as a tree, according to the depth-first search order, by the index set $\{1, \cdots, 6\}$. Then, we have that $\phi(6) = E(x, y)$, $\phi(5) = \exists x \phi(6)$, $\phi(4) = E(x, y)$, $\phi(3) = \phi(4) \wedge \phi(5), \phi(2) = \forall y \phi(3)$, and $\phi(1) = \exists x \phi(2)$.

We say that an index i is a *parent* of an index j, and also that j is a *child* of i, if, in viewing the formula ϕ as a tree, the root of the subformula occurrence of i is the parent of the root of the subformula occurrence of j. Note that, when this holds, the formula $\phi(i)$ either is of the form $Qv\phi(j)$ where Q is a quantifier and v is a variable, or is a conjunction where $\phi(j)$ appears as a conjunct. For example, with respect to the sentence and indexing in Example 2, index 3 has two children whose index are 4 and 5, and index 3 has one parent whose index is 2.

Definition 1 (Judgement) Let $(\phi, \mathbf{B})$ be a QCSP instance. A *constraint* on $(\phi, \mathbf{B})$ is a pair (V, F) where V is a set of variables occurring in ϕ, and F is a set of mappings from V to B. A *judgement* on $(\phi, \mathbf{B})$ is a triple (i, V, F) where $i \in I_\phi$ and (V, F) is a constraint with $V \subseteq \mathsf{freeVar}(\phi(i))$; it is *empty* if $F = \emptyset$.

Roughly speaking, the role of a judgement (i, V, F) on $(\phi, \mathbf{B})$ is to collect in F the mappings f on the variables V that are "candidates" to satisfy $\mathbf{B}, f \models \phi(i)$. The construction of judgements is based on operations over mappings (from variables to elements of the domain) and sets of mappings. When $\mathbf{B}$ is a structure, ϕ is a formula over the vocabulary of $\mathbf{B}$ and f is a mapping from the free variables of ϕ to the universe of $\mathbf{B}$, we write $\mathbf{B}, f \models \phi$ to indicate that ϕ is satisfied in $\mathbf{B}$ under f.When f is a mapping and $z \in B$, we use $f[x \mapsto v]$ to denote the extension or update of f that maps x to v. This notation is also used multiple updating as $f[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ and also $f[X \mapsto V]$ where X, V respectively represents the tuples $(x_1 \ldots, x_n)$ and $(v_1 \ldots, v_n)$. When f is mapping from V to B and U is a subset of V, we use $f \upharpoonright U$ to denote the restriction of f to U.

Definition 2 (Operations over sets of mappings) Let (U_1, F_1), (U_2, F_2) be two constraints on the same QCSP instance, we define the *join* of F_1 and F_2, denoted by $F_1 \bowtie F_2$, to be

$$F_1 \bowtie F_2 = \{f : U_1 \cup U_2 \to B \mid (f \upharpoonright U_1) \in F_1, (f \upharpoonright U_2) \in F_2\}.$$

Let (V, F) be a constraint and $U \subseteq V$ with $\{w_1, w_2, \ldots, w_r\} = V \setminus U$, we define the *projection* and the *dual-projection* of F on U, respectively denoted by $F \restriction U$ and $F\# U$, to be

$$F \restriction U = \{f \restriction U : U \to B \mid f \in F\}$$

$$F\# U = \{f : U \to B \mid f[w_1 \mapsto b_1, \ldots, w_r \mapsto b_r] \in F \text{ for all } b_1, b_2, \ldots, b_r \in B\}.$$

The *dual-projection* is used to deal with universally quantified variables. Dually, *projection* can be used to cope with existential quantification. We adopt the convention that (relative to a QCSP instance) there is exactly one map $e : \emptyset \to B$ defined on the empty set, so there are two constraints whose variable set is the empty set: the constraint $(\emptyset, \emptyset)$, and the constraint $(\emptyset, \{e\})$ where e is the aforementioned map.

The proof system PS is refutation-based in the sense that it aims to find a proof of the empty judgement $(-, \emptyset, \emptyset)$ on $(\phi, \mathbf{B})$, which means that $\mathbf{B} \not\models \phi$. The next definition introduces the inference rules of the proof system PS, as it was defined in [1].

Definition 3 **(PS proof system)** A *judgement proof* on a QCSP-instance $(\phi, \mathbf{B})$ on signature σ is a finite sequence of judgements, each of which is obtained by the application of the following inference rules:

$$\text{(atom)} \quad \frac{}{(i, V, F)} \quad \text{where} \begin{cases} R \in \sigma \text{ such that } \mathsf{ar}(R) = k \\ V = \{v_1, \cdots, v_k\} \\ \phi(i) = R(V) \\ F = \{f : V \to B \mid (f(v_1), ..., f(v_k)) \in R^{\mathbf{B}}\} \end{cases}$$

$$\text{(join)} \quad \frac{(i, U_1, F_1) \quad (i, U_2, F_2)}{(i, U_1 \cup U_2, F_1 \bowtie F_2)}$$

$$\text{(projection)} \quad \frac{(i, V, F)}{(i, U, F \restriction U)} \quad \text{where } U \subseteq V$$

$$(\forall\text{-elimination}) \quad \frac{(j, V, F)}{(i, V \setminus \{y\}, F\# (V \setminus \{y\}))} \quad \text{where} \begin{cases} y \in V \\ \phi(i) = \forall y \phi(j) \\ i \text{ is the parent of } j \end{cases}$$

$$\text{(upward flow)} \quad \frac{(j, V, F)}{(i, V, F)} \quad \text{where } i \text{ is the parent of } j$$

Given an instance $(\phi, \mathbf{B})$, we say that a judgement (i, V, F) is *derivable* on $(\phi, \mathbf{B})$ if there exists a judgement proof on $(\phi, \mathbf{B})$ that contains (i, V, F).

It is worthy noting that Definition 1 requires of a triple (i, V, F), to be a judgement, that all variables in V must be free variables of $\phi(i)$. Consequently, since PS only deals with judgements, the (upward flow) rule can only be applied to a judgement (j, V, F) if all variables in V are free variables of $\phi(i)$, where i is the parent of j.

In Example (3) we present a judgement proof according to PS, where $(1, \emptyset, \emptyset)$ is derived. It also shows how the combination of both the upward flow rule and the projection rule derives judgements where the number of variables is minimum.

Obviously, the correctness of the upward flow rule relies in the fact that CPL logical symbols $(\forall, \exists, \wedge)$ are 'positive'.

Example 3 Let ϕ be the sentence from Example 2 over signature $\sigma=\{E\}$ with $ar(E) = 2$. Consider ϕ to be indexed as shown in Figure 1, where $\phi(6) = E(x, y)$, $\phi(5) = \exists x\phi(6)$, $\phi(4) = E(x, y)$, $\phi(3) = \phi(4) \wedge \phi(5), \phi(2) = \forall y\phi(3)$, and $\phi(1) = \exists x\phi(2)$. Let $\mathbf{B}$ be the structure over σ with domain $B = \{a, b, c\}$ such that $E^{\mathbf{B}} = \{(a, a), (a, c), (b, a)\}$. Let G_E be the set of mappings from $\{x, y\}$ to B that satisfy $E(x, y)$ (over $\mathbf{B}$):

$$G_E = \{\{x \mapsto a, y \mapsto a\}, \{x \mapsto a, y \mapsto c\}, \{x \mapsto b, y \mapsto a\}\}.$$

A possible judgement proof on $(\phi, \mathbf{B})$ is the following.

(1)– (atom):	$(6, \{x, y\}, G_E)$
(2)– From (1) by (projection):	$(6, \{y\}, \{\{y \mapsto a\}, \{y \mapsto c\}\})$
(3)– From (2) by (upward flow):	$(5, \{y\}, \{\{y \mapsto a\}, \{y \mapsto c\}\})$
(4)– From (3) by (upward flow):	$(3, \{y\}, \{\{y \mapsto a\}, \{y \mapsto c\}\})$
(5)– From (4) by ($\forall$-elimination):	$(2, \emptyset, \emptyset)$
(6)– From (5) by (upward flow):	$(1, \emptyset, \emptyset)$

The work presented in this paper is based on viewing the set of statements that can be derived by a proof system as the least fixpoint of the derivability relation that is induced by the set of inference rules of the considered proof system. Next, we illustrate this view of the proof system PS to provide a good basis of the general theory underlying our formalization.

The set of all judgements that are derivable on a given QCSP instance $(\phi, \mathbf{B})$ can be seen as a least fixpoint of an operator that we call $D_{(\phi,\mathbf{B})}$. Next, we formally define this operator. Let $\mathcal{J}$ be the set of all derived judgements on a QCSP instance $(\phi, \mathbf{B})$. The set $\mathcal{P}(\mathcal{J})$ (all subsets over $\mathcal{J}$) is a partial ordered defined by the $\subseteq$-relation among sets. $D_{(\phi,\mathbf{B})}$ is a map from $\mathcal{P}(\mathcal{J})$ to $\mathcal{P}(\mathcal{J})$ which, given any $S \in \mathcal{P}(\mathcal{J})$, is defined as

$$\begin{aligned} D_{(\phi,\mathbf{B})}(S) = S \cup \{\, j \in \mathcal{J} \mid{} & j \text{ is obtained by applying one of the inference} \\ & \text{rules to a judgement } s \in S \,\}. \end{aligned}$$

Given any QCSP instance $(\phi, \mathbf{B})$, the least fixpoint of $D_{(\phi,\mathbf{B})}$ is the set of all derivable judgementes according to the fixpoint semantics.

Example 4 Let ϕ be the sentence $\phi = \exists x\forall y P(x, y)$ where $\phi(3) = P(x, y)$; $\phi(2) = \forall y\phi(3)$; $\phi(1) = \exists x\phi(2)$. Consider ϕ as a sentence over signature $\{P\}$ with $ar(P) = 2$. Define $\mathbf{B}$ to be a structure over this signature having domain $B = \{a, b, c\}$ and where $P^{\mathbf{B}} = \{(a, a), (a, b)\}$

Let F_P be the set of mappings from $\{x, y\}$ to B that satisfy $P(x, y)$ (over $\mathbf{B}$):

$$F_P = \{\{x \mapsto a, y \mapsto a\}, \{x \mapsto a, y \mapsto b\}\}.$$

For this QCSP instance we calculate the fixpoint of $D_{(\phi,\mathbf{B})}$.

$$\begin{array}{lll} D_{(\phi,\mathbf{B})} \uparrow 0 & = & \emptyset \\ D_{(\phi,\mathbf{B})} \uparrow 1 & = & D_{(\phi,\mathbf{B})}(\emptyset) \quad = \quad \{(3, \{x,y\}, F_P)\} \\ D_{(\phi,\mathbf{B})} \uparrow 2 & = & D_{(\phi,\mathbf{B})}(D_{(\phi,\mathbf{B})} \uparrow 1) \end{array}$$
$$= D_{(\phi,\mathbf{B})} \uparrow 1 \cup \{(2, \{x\}, \emptyset), (3, \{x\}, (F_P \upharpoonright \{x\})), (3, \{y\}, (F_P \upharpoonright \{y\})), (3, \emptyset, \{e\})\}$$
$$\begin{array}{lll} D_{(\phi,\mathbf{B})} \uparrow 3 & = & D_{(\phi,\mathbf{B})} \uparrow 2 \cup \{(2, \emptyset, \emptyset), (2, \{x\}, (F_P \upharpoonright \{x\})), (2, \emptyset, \{e\})\} \\ D_{(\phi,\mathbf{B})} \uparrow 4 & = & D_{(\phi,\mathbf{B})} \uparrow 3 \cup \{(1, \emptyset, \emptyset), (1, \emptyset, \{e\})\} \\ D_{(\phi,\mathbf{B})} \uparrow 5 & = & D_{(\phi,\mathbf{B})}(D_{(\phi,\mathbf{B})} \uparrow 4) \quad = \quad D_{(\phi,\mathbf{B})} \uparrow 4 \text{ which is the least fixpoint.} \end{array}$$

Therefore, the empty judgement $(1, \emptyset, \emptyset)$ belongs to the least fixpoint of the derivability relation associated to the studied QCSP-instance.

Example 5 Let ϕ be the sentence from Example 2 over signature σ=$\{E\}$ with $ar(E) = 2$. Consider ϕ to be indexed as shown in Figure 1, where $\phi(6) = E(x, y)$, $\phi(5) = \exists x\phi(6)$, $\phi(4) = E(x, y)$, $\phi(3) = \phi(4) \wedge \phi(5), \phi(2) = \forall y\phi(3)$, and $\phi(1) = \exists x\phi(2)$. Let $\mathbf{B}$ be the structure over σ with domain $B = \{a, b, c\}$ such that $E^{\mathbf{B}} = \{(a, a), (a, b), (a, c), (b, a)\}$. Let F_E be the set of mappings from $\{x, y\}$ to B that satisfy $E(x, y)$ (over $\mathbf{B}$):

$$F_E = \{\{x \mapsto a, y \mapsto a\}, \{x \mapsto a, y \mapsto b\}, \{x \mapsto a, y \mapsto c\}, \{x \mapsto b, y \mapsto a\}\}.$$

The least fixpoint of $D_{(\phi,\mathbf{B})}$ is calculated below, where K is the set of mappings $\{\{x \mapsto a\}, \{x \mapsto b\}\}$.

$$\begin{aligned} &D_{(\phi,\mathbf{B})} \uparrow 0 = \emptyset \\ &D_{(\phi,\mathbf{B})} \uparrow 1 = \{(4, \{x,y\}, F_E), (6, \{x,y\}, F_E)\} \\ &D_{(\phi,\mathbf{B})} \uparrow 2 = D_{(\phi,\mathbf{B})} \uparrow 1 \cup \{(4, \{y\}, H), (4, \emptyset, \{e\}), (4, \{x\}, K), \\ &\qquad\qquad (6, \{y\}, H), (6, \emptyset, \{e\}), (6, \{x\}, K), (3, \{x,y\}, F_E)\} \\ &D_{(\phi,\mathbf{B})} \uparrow 3 = D_{(\phi,\mathbf{B})} \uparrow 2 \cup \{(3, \{y\}, H), (3, \emptyset, \{e\}), (3, \{x\}, K), \\ &\qquad\qquad (5, \{y\}, H), (5, \emptyset, \{e\}), (2, \{x\}, G)\} \\ &D_{(\phi,\mathbf{B})} \uparrow 4 = D_{(\phi,\mathbf{B})} \uparrow 3 \cup \{(2, \emptyset, \{e\}), (2, \{x\}, K)\} \\ &D_{(\phi,\mathbf{B})} \uparrow 5 = D_{(\phi,\mathbf{B})} \uparrow 4 \cup \{(1, \emptyset, \{e\})\} \\ &D_{(\phi,\mathbf{B})} \uparrow 6 = D_{(\phi,\mathbf{B})} \uparrow 5 \text{ which is the least fixpoint.} \end{aligned}$$

Hence, the empty judgement $(1, \emptyset, \emptyset)$ is not in the fixpoint of $D_{(\phi,\mathbf{B})}$. This means that the empty judgement cannot appear in any judgement proof on the considered QCSP-instance.

It is obvious, by construction, that the least fixpoint of $D_{(\phi,\mathbf{B})}$ is the set of all judgements that are derivable on $(\phi, \mathbf{B})$. Consequently, metalogical properties of the set of all judgements that are derivable on $(\phi, \mathbf{B})$ can be proved by induction on the number of iterations of the operator $D_{(\phi,\mathbf{B})}$. By Tarski's Theorem [48], the existence of the least fixpoint of the operator $D_{(\phi,\mathbf{B})}$ (over the boolean lattice) requires $D_{(\phi,\mathbf{B})}$ to be monotonic, hence such fact should be also ensured to validate

any inductive proof on the number of iterations.
The next theorem establishes the correctness and completeness of PS. Its proof has been made in Dafny on the basis of the least fixpoint semantics, and it is one of the main contribution of this work.

Theorem 1 (Correctness and Completeness of PS) *Let $(\phi, \mathbf{B})$ be a QCSP instance. Assume the root of ϕ has index r. The empty judgement $(r, \emptyset, \emptyset)$ is derivable if and only if $\mathbf{B} \not\models \phi$.*

In Example 4, the empty judgement $(1, \emptyset, \emptyset)$ is derivable. Therefore, by Theorem 1, $\mathbf{B} \not\models \phi$. In Example 5, the empty judgement $(1, \emptyset, \emptyset)$ is not in the least fixpoint of $D_{(\phi,\mathbf{B})}$, by Theorem 1, it holds that $\mathbf{B} \models \phi$.

3 Dafny: Language, Verifier and IDE

Dafny [28] is a program verifier that includes a programming language and specification constructs. The Dafny user creates and verifies both specifications and implementations. Dafny specification language extends first-order logic with algebraic data types, extreme/inductive predicates, induction (also co-induction), generic types, abstracting and refining modules, assertions and many others built-in specification features that makes Dafny a good candidate for our work. In this section, we briefly introduce the main notions of Dafny that facilitates the understanding of the rest of the paper.

The basic unit of a Dafny program is the `method`. A method is a piece of executable code with a head where multiple named parameters and multiple named results are declared. Dafny has also built-in specification constructs for assertions, such as `requires` for preconditions, `ensures` for postconditions, and `assert` for inline assertions. Using `requires` and `ensures` we specify methods and lemmas. Assertions specify properties that are satisfied at some point. Assertions are mainly used to provide hints to the verifier. In other words, once the assertion is proved, it turns into a usable property for completing the proof. Indeed, "`assert` φ" tells Dafny to check that φ holds and to use the condition φ (as a lemma) to prove the properties beyond this point. Dafny distinguishes between *ghost* entities and *executable* entities. Ghost entities are used only during verification; the compiler omits them from the executable code. The `lemma` declarations are like methods, but no code is generated for them, i.e. a lemma is equivalent to ghost method. The body of a lemma is its proof. Dafny also offers user-defined specification constructs (which are ghost), such as `function` and `predicate` that can be defined by well-founded inductive definitions, built-in immutable types, polymorphic (inductive and coinductive) algebraic `datatype`s, inductive and co-inductive `predicate`s. Dafny also provides built-in immutable type, such as `set`, `multiset`, `map` and `seq` –which respectively denote the types of finite sets, multisets, maps, tuples, and sequences– that are very useful in specification. These built-in types are equipped with the usual operations, including set comprehension expressions:

```
set x1 : T1, x2:T2, ... | P(x1,x2,...) • E(x1,x2,...)
```

for defining the set of all values given by the expression `E(x1,x2,...)` for all finite tuples `(x1,x2,...)` such that `P(x1,x2,...)`.[1] For lemma proofs, Dafny provides a special notation that is easy to read and understand: *calculations* [30]. A calculation in Dafny is a statement that proves a property. This notation was extracted from the *calculational method* [2], whereby a theorem is established by a chain of formulas, each transformed in some way into the next. The relationship between successive formulas (for example, equality, implication, double implication, etc.) is notated, or it can be omitted if it is the default relationship (equality). In addition, the hints (usually asserts or lemma calls) that justify a step can also be notated (in curly brackets after the relationship). Calculations are written inside the environment `calc{ }`.

The Dafny specification constructor `inductive predicate` (also called *extreme predicates*) [33] allows the definition of a predicate as an extreme solution: a least fixpoint of a set of recursive rules.[2] Inductive predicates are essential to formally define the set of judgments that can be proved by the proof system PS (introduced in the previous section). Properties of inductive predicates can be proved by induction in the construction of the least fixpoint of an inductive predicate `P(x)`. Such properties must be coded as `inductive lemma`s for least fixpoint. Dafny offers a standard way to set up the proof of these kind of lemmas, by induction on the number of iterations of the operator whose least fixpoint is the meaning of `P(x)`. To validate such inductive proofs, according to Tarski's Theorem [48], Dafny verifies the monotonicity of `P`, by checking out that every call to `P` (in its definition) is under an even number of negations. Very detailed and helpful explanations on inductive predicates and inductive lemmas can be found in [33]. In Section 4 we introduce an inductive predicate (`is_derivable`) and prove an inductive lemma (`models_Lemma`).

The Dafny integrated development environment (IDE) is an extension of Microsoft Visual Studio (VS). The IDE is designed to reduce the effort required by the user to make use of the system. The IDE runs the program verifier in the background and provides design time feedback. Assertions are sent to the SMT solver Z3 (a fully automatic theorem prover) to check its satisfiability that will be reported to the Dafny user. Assertion violations in lemma proofs, as well as verification errors, are reported along with different informations such as the locations (of the properties) related to the error. The interested reader is referred to [31] for further information on the several ways that Dafny IDE helps to build both lemma proofs and verified software. Dafny is able to export executable files (.exe), libraries (.dll) and .Net source code (.cs) with the implementation of the functionality specified, whenever the automatic verification is successful and every lemma is proved.

[1] For easy reading, in the Dafny code snippets, we show the usual mathematical symbols, instead of real Dafny notation. For example, we show $\bullet$ for ::(such that), $\cup$ for union instead of `+`, $\subseteq$ for set inclusion instead of `<=`, also for the logical symbols and quantifiers, for example `&&` is shown as $\wedge$ and `forall` as $\forall$, etc.

[2] Dafny also provides co-induction based on greatest fixpoints (see [33]), but they are not used in this paper.

4 Formalization of the Proof System *PS* in Dafny

In this section we explain the main types and definitions that make up our formalization. We first formalize what are (well-formed) structures, formulas, QCSP-Instances and judgements. Then, we define the operations on judgements and the inductive predicate that formalizes the derivability relation of *PS*.

Structures A structure is given by a triple formed by a signature, a domain (i.e. a non-empty finite set), and an interpretation, which is a map from the names in the signature to relations on the domain of the arity determined in the signature.

```
type Name = string

type Signature =  map<Name,int>

type Interpret<T> = map<Name,set<seq<T>>>

datatype Structure<T> =
         Structure(Sig:Signature,Dom:set<T>,I:Interpret<T>)

predicate wfStructure<T>(B:Structure<T>)
{
B.Dom ≠ {} ∧
∀ r • r in B.Sig.Keys ⟹ (r in B.I ∧
                          ∀ t • t in B.I[r] ⟹ |t| = B.Sig[r])
}
```

The variable of type `T` represents the type of the elements in the domain, relations in the domain are represented by the set of sequences (viewed as tuples) that belongs to the relation. Hence, the predicate `wfStructure` decides the non-emptiness of the domain along with every relation symbol is interpreted by sequences whose length is its arity.

Formulas and QCSP-instances We define the syntax of Conjunctive Positive Logic formulas as a datatype, where for example an atom $R(x1, x2, x3)$ is represented as `Atom("R",[x1,x2,x3])`. In the datatype `Formula` each constructor has two destructors giving access to each component of the formula.

```
datatype Formula =   Atom(rel:Name, par:seq<Name>)
                   | And(0:Formula, 1:Formula)
                   | Forall(x:Name, Body:Formula)
                   | Exists(x:Name, Body:Formula)

predicate wfFormula(S:Signature, phi:Formula)
{
match phi
   case Atom(R, par) => R in S.Keys ∧ |par| = S[R]
   case And(phi0, phi1) => wfFormula(S, phi0) ∧ wfFormula(S, phi1)
   case Forall(x, alpha) => wfFormula(S,alpha)
   case Exists(x, alpha) => wfFormula(S,alpha)
}

function freeVar(phi:Formula): set<Name>
{
match phi
   case Atom(R, par) => setOf(par)
   case And(ph1, phi1) => freeVar(ph1) + freeVar(phi1)
```

```
    case Forall(x, phi) => freeVar(phi) - {x}
    case Exists(x, phi) => freeVar(phi) - {x}
}

predicate sentence(phi:Formula) { freeVar(phi) = {} }
```

The predicate `wfFormula` decides whether a formula is well-formed with respect to a given signature, that is if the number of parameters of all its atoms coincides with its arity. The function `freeVar` gives the set of its free variables, and the predicate `sentence` decides whether a formula has no free variables.

```
predicate wfQCSP_Instance(phi:Formula, B:Structure)
{
wfStructure(B) ∧ wfFormula(B.Sig,phi) ∧ sentence(phi)
}
```

A weel-formed QCSP-instance consist of a well-formed structure, a well-formed formula with symbols in the signature of the structure that must be a sentence. For example, if `phi` is `Exists(x,Forall(y,And(Atom(E,[x, y]),Exists(x,Atom(E,[x,y])))))` and `B` is `Structure(map[E↦2],set`$\{a,b,c\}$`,map[E ↦ set`$\{[a,a],[a,b],[a,c],[b,a]\}$`])` which represents the QCSP-intance of Example 3, then `wfQCSP_Instance(phi, B)` is `True`.

Judgements We also declare judgements and the predicate for checking its well-formedness as follows: [3]

```
type Valuation<T> = map<Name, T>

datatype Judgement<T> = J(i:Index,V:set<Name>,F:set<Valuation<T>>)

predicate wfJudgement<T>(j:Judgement<T>, phi:Formula, B:Structure<T>)
requires wfQCSP_Instance(phi,B)
{
j.i in setOfIndex(phi)                ∧
j.V ⊆ freeVar(FoI(j.i,phi,B.Sig))    ∧
(∀ f • f in j.F ⟹ j.V = f.Keys)     ∧
(∀ f, v • f in j.F ⟹ v in f.Values ⟹ v in B.Dom)
}
```

A (well-formed) judgement on a (well-formed) QCSP-instance $(\phi, \mathbf{B})$, as defined in Section 2, is a triple formed by an index `i` on the set of index of ϕ, a set of variables included in the free variables of the subformula of index `i` of ϕ and a set of maps from exactly these variables to elements of the domain of the structure. For that, `setOfIndex` is a function that computes the set of indexes in the nodes of a given formula (seen as a tree, see Figure 1). In our formalization, for easy access to formula nodes, indexes are sequences of zeros and ones, instead of natural numbers. We do not explain here the technical details of that formalization. Given an index `i`, a formula `phi` and a signature `S`, the function called `FoI(i, phi, S)` returns the subformula of `phi` of index `i`. The parameter `S` is added for expressing that the function `FoI` preserves the well-formedness property with respect to the signature of `phi`.

Operations on judgements The inference rules in *PS* rely upon apply the operations join, projection and dual-projection on the sets of valuations, which are

[3] In Dafny code, one line comments start by // and are coloured in green.

part of one or two judgements (the component `F` in the datatype) associated to a QCSP-instance given by a formula ϕ and a structure B. We define (in Dafny) the following predicates on judgements to decide whether a judgement is a projection or a dual-projection of another judgement, and also whether a judgement is the joint of two given judgements.

```
predicate is_projection<T> (j1: Judgement<T>, j2: Judgement<T>,
                            phi: Formula, B: Structure<T>)
// j1 is a projection of j2
requires wfJudgement(j1,phi,B) ∧ wfJudgement(j2,phi,B)
{
j1.i = j2.i ∧ j1.V ⊆ j2.V ∧
j1.F = ( set f | f in j2.F • projectVal(f,j1.V) )
}

predicate is_dualProjection<T> (j1: Judgement<T>, v: Name,
                                j2: Judgement<T>,
                                phi: Formula, B: Structure<T>)
// j1 is a dual projection of j2 (on variable v)
requires wfJudgement(j1,phi,B) ∧ wfJudgement(j2,phi,B)
{
j2.i = j1.i + [0] ∧ j1.V = j2.V - {v} ∧ v in j2.V ∧
j1.F = (set h: Valuation<T> | h in allMaps(j1.V, B.Dom) ∧
                              ∀ b • b in B.Dom ⟹ h[v:=b] in j2.F)
}

predicate is_join<T> (j: Judgement<T>, j1: Judgement<T>,
                      j2: Judgement<T>, phi: Formula, B: Structure<T>)
// j is the join of j1 and j2
requires wfJudgement(j,phi,B)
requires wfJudgement(j1,phi,B) ∧ wfJudgement(j2,phi,B)
{
j.i = j1.i = j2.i ∧ j.V = j1.V + j2.V ∧
j.F = (set f: Valuation<T> | f in allMaps(j1.V+j2.V, B.Dom) ∧
                            projectVal(f,j1.V) in j1.F  ∧
                            projectVal(f,j2.V) in j2.F)
}
```

For a judgement `j`, the expression `j.i` is the index in the tree that represents the formula, and `j.i + [0]` (respectively `j.i + [1]`) is the index of its left-hand (resp. rigth-hand) child. If it has only one child, it is `j.i + [0]`. Function `projectVal`, when applied to any `f: Valuation<T>` and any `U: set<Name>)` such that `U` ⊆ `f.Keys`, calculates `(map s | s in U • f[s]): Valuation<T>`, hence `projectVal(f,U).Keys =U` is ensured. In the above predicate `is_join`, Dafny checks the finiteness of the set `allMaps(j1.V+j2.V, B.Dom)`. Indeed, Dafny checks the finiteness of `X` for any expression `x in X` where `X` is a set. Function `allMaps` is applied to two parameters `keys: set<A>` and `values: set<B>)`. Function `allMaps` gives the set of all maps whose domain is `keys` and whose range is a subset of `values`. Indeed, we prove this fact in lemma `allMaps_Correct_Lemma`.

The above three predicates `is_join`, `is_projection` and `is_dualProjection`, along with the following predicate `is_upwardFlow` respectively enable the encoding of the inference rule (join), (projection), (∀-elimination) and (upward flow), which are given in Definition 3.

```
predicate is_upwardFlow<T> (j1: Judgement<T>, j2: Judgement<T>,
                            phi: Formula, B: Structure<T>)
// j1 is the upwardFlow of j2
```

```
requires wfJudgement(j1,phi,B) ∧ wfJudgement(j2,phi,B)
{
j2.V = j1.V ∧ j2.F = j1.F ∧
(
(FoI(j1.i,phi,B.Sig).And? ∧ (j2.i = j1.i+[0] ∨ j2.i = j1.i+[1]))
∨
((FoI(j1.i,phi,B.Sig).Forall? ∨ FoI(j1.i,phi,B.Sig).Exists?) ∧
  j2.i=j1.i+[0])
)
}
```

Derivability predicate The following inductive predicate `is_derivable` defines, in a very natural way, the least fixpoint of the derivabilty relation induced by the five rules in Definition 3.

```
inductive predicate is_derivable<T(!new)> (j:Judgement<T>,
                                           phi:Formula,
                                           B:Structure<T>)
requires wfQCSP_Instance(phi,B) ∧ wfJudgement(j,phi,B)
{
var phii := FoI(j.i,phi,B.Sig);
( // rule (atom)
phii.Atom?
∧ j.V = setOf(phii.par)
∧ j.F = (set f:Valuation<T> | f in allMaps(j.V, B.Dom)
                              ∧ HOmap(f,phii.par) in B.I[phii.rel])
) ∨ ( // rule (projection)
∃ j' • wfJudgement(j',phi,B) ∧ is_projection(j,j',phi,B)
             ∧ is_derivable(j',phi,B)
) ∨ ( // rule (join)
phii.And?
∧ ∃ j0,j1 • wfJudgement(j0,phi,B)  ∧ wfJudgement(j1,phi,B)
             ∧ j0.i = j1.i
             ∧ is_join(j,j0,j1,phi,B)
             ∧ is_derivable(j0,phi,B)  ∧ is_derivable(j1,phi,B)
) ∨ ( // rule (∀-elimination)
phii.Forall?
∧ ∃ j' • wfJudgement(j',phi,B)
         ∧ phii=Forall(phii.x,FoI(j'.i,phi,B.Sig))
         ∧ is_dualProjection(j,phii.x,j',phi,B)
         ∧ is_derivable(j',phi,B)
) ∨ ( // rule (upward flow)
∃ j' • wfJudgement(j',phi,B)
          ∧ is_upwardFlow(j,j',phi,B) ∧ is_derivable(j',phi,B)
)
}
```

In Dafny, inductive predicate definitions are not allowed to depend on the allocation state. The suffix `(!new)`, on parameter type `T` (it is shown as super-index in the Dafny code snippets), restricts the instances of `T` to types that do not contain any reference to an object (or pointer), and thus does not depend on the allocation state. This is a quite recently added type-parameter characteristic `(!new)`, in the same vein as the suffix `(==)` restricts instances to be equality-supporting types.

In the encoding of the rule (atom), we use the auxiliary function `HOmap` for applying the function `f` to the list of arguments `phii.par`, this gives a tuple that is checked to belong to the interpretation of relation `phii.R` in the structure `B`.

5 Dafny Proofs of Soundness and Completeness

In this section we explain the main ingredients of the Dafny proof for Theorem 1, which ensures that PS is a sound and complete proof system for QCSP instances. The forward direction of Theorem 1 states the soundness result that is proved in Dafny lemma `soundness_Theorem`. The backward direction is the completeness statement that is proved by the Dafny lemma `completeness_Theorem`. For expressing these meta-logical results we use the following predicate that states whether a QCSP-instance `(B,f)` is a model of a formula `phi`.

```
predicate models<T>(B: Structure<T>, f: Valuation<T>, phi: Formula)
requires wfStructure(B) ∧ wfFormula(B.Sig,phi) ∧ f.Values ⊆ B.Dom
decreases phi
{
(freeVar(phi) ⊆ f.Keys) ∧
match phi
  case Atom(R,par) => HOmap(f,par) in B.I[R]
  case And(phi0,phi1) => models(B,f,phi0) ∧ models(B,f,phi1)
  case Forall(x,alpha) => ∀ v • v in B.Dom
                              ⟹ models(B,f[x:=v],alpha)
  case Exists(x,alpha) => ∃ v • v in B.Dom
                              ∧ models(B,f[x:=v],alpha)
}
```

The above three recursive cases are obvious. The case `Atom(R,par)`, using the auxiliary function `HOmap`, applies the function `f` to the list of arguments `par`, and checks if the resulting tuple belongs to the interpretation of relation `R` in the structure `B`.

On the basis of the above predicate `models`, we define:

```
predicate valuationModel<T> (h: Valuation<T>, j: Judgement<T>,
                             phi: Formula, B: Structure<T>)
{
h in allMaps(j.V, B.Dom) ∧
wfStructure(B) ∧ wfFormula(B.Sig,phi) ∧ wfJudgement(j,phi,B) ∧
models(B,h,existSq(freeVar(FoI(j.i,phi,B.Sig))-j.V,
                   FoI(j.i,phi,B.Sig)))
}
```

Given a valuation `h` and a judgement `j` on a QCSP-instance `(phi,B)`, predicate `valuationModel` decides whether `(B,h)` models the subformula of `phi` given by the index of `j.i` properly closed with existential quantifiers on all the variables that do not belong to `j.V`. The expression `FoI(j.i,phi,B.Sig))` represents the formula $\phi(i)$, the expression `freeVar(FoI(j.i,phi,B.Sig))-j.V` represents $\mathsf{freeVar}(\phi(i))\backslash$`j.V` and the function `existSq` enables the existential closing (for its definition see Figure 3 at the end of Section 6). Hence, the Dafny expression

```
existSq(freeVar(FoI(j.i,phi,B.Sig))-j.V,FoI(j.i,phi,B.Sig))
```

encodes the formula $\exists x_1 \ldots \exists x_n \phi(i)$ provided that $\mathsf{freeVar}(\phi(i))\backslash$`j.V` $= \{x_1, \ldots, x_n\}$.

The `soundness_Theorem` will be prove as an easy consequence of the following:

Lemma 1 *Let $(\phi, \mathbf{B})$ be a QCSP instance and (i, V, F) a derivable judgement (on it). Let $\{v_1, v_2, \ldots v_n\}$ be the variables in $\mathsf{freeVar}(\phi(i)) \setminus V$. For all $h : V \to B$ it holds that $\mathbf{B}, h \models \exists v_1 \ldots \exists v_n \phi(i)$ implies $h \in F$.*

which is encoded in Dafny as the following inductive lemma, whose proof is partially shown:

```
inductive lemma models_Lemma<T> (j:Judgement<T>, phi:Formula,
                                 B:Structure<T>)
requires wfQCSP_Instance(phi,B) ∧ wfJudgement(j,phi,B)
requires is_derivable(j,phi,B)
ensures ∀ h • valuationModel(h,j,phi,B) ⟹ h in j.F
{
var phii := FoI(j.i, phi,B.Sig);
if phii.Atom? ∧ j.V = setOf(phii.par)
   ∧ j.F = (set f: Valuation<T> | f in allMaps(j.V,B.Dom)
                             ∧ HOmap(f,phii.par) in B.I[phii.rel])
{// (atom)
 ∀ h | valuationModel(h,j,phi,B) {allMaps_Correct_Lemma(h,B.Dom);}
}
else if ∃ j' • wfJudgement(j',phi,B) ∧ is_projection(j,j',phi,B)
                                    ∧ is_derivable(j',phi,B)
{// (projection)
 var j' :| wfJudgement(j',phi,B) ∧ is_projection(j,j',phi,B)
                               ∧ is_derivable(j',phi,B);
 models_Lemma(j',phi,B);
 projection_Lemma(j,j',phi,B);
}
else if ... {// (join)
}
else if ... {// (∀-elimination)
}
else {// (upward flow)
}
}
```

Since `models_Lemma` is an inductive lemma with hypothesis `is_derivable(j,phi,B)`, the proof makes induction in the construction of the inductive predicate `is_derivable`, the inductive proof of `models_Lemma` has a base case for the rule (atom) and one inductive case for each of the remaining four rules. In the base case (atom), for all valuation h such that $\mathbf{B}, h \models \exists v_1 \ldots \exists v_n \phi(i)$, we call the auxiliary lemma `allMaps_Correct_Lemma` to show that the set `allMaps(h, B.Dom)` really contains all the maps with domain in `h.Keys` that give values in `B.Dom`. In the inductive case where the judgement `j` is a projection of another derivable judgement `j'`, we recursively call `models_Lemma(j',phi,B)` for the induction hypothesis that ensures that all (valuations that are) models of `j'` are in `j'.F`. Then, the call `projection_Lemma(j,j',phi,B)` invokes the following auxiliary lemma:

```
lemma projection_Lemma<T>(j:Judgement<T>, j':Judgement<T>,
                          phi:Formula, B:Structure<T>)
requires wfStructure(B) ∧ wfFormula(B.Sig,phi)
requires wfJudgement(j,phi,B) ∧ wfJudgement(j',phi,B)
requires is_projection(j,j',phi,B) // (H1)
requires ∀ h • valuationModel(h,j',phi,B)  ⟹ h in j'.F //(H2)
ensures ∀ h • valuationModel(h,j,phi,B) ⟹ h in j.F
```

The lemma `projection_Lemma` assumes, the well-formedness of all its parameters along with, the hypothesis `(H1)`: `j` is the projection of `j'` and the fact (as hypothesis `(H2)`) that `j'` satisfies the ensures of lemma `models`, then it ensures that also all valuations that are models of `j` belongs to `j.F`. Next, we explain the Dafny proof of `projection_Lemma` whose code is:[4]

[4] We use to include commented assertions whenever Dafny does not need them as hints, but serve as documentation to the reader, who could check its validity by uncommenting them.

```
var phii := FoI(j.i,phi,B.Sig);
var W := freeVar(phii)-j'.V;
var Y := j'.V - j.V;
var X := freeVar(phii)-j.V;
∀ h:Valuation<T> | valuationModel(h,j,phi,B)
        ensures h in j.F;
        {
        //assert  models(B,h,existSq(X,phii));
        assert X == Y+W;
        //assert models(B,h,existSq(Y+W,phii));
        existSq_Sum_Lemma(B, h, Y, W, phii);
        assert models(B,h,existSq(Y,existSq(W,phii)));
        existSqSem_Lemma(B, h, Y, existSq(W,phii));
        var U, Z :| setOf(Z) ≤ B.Dom ∧ |U| == |Z| == |Y|
                    ∧ setOf(U) == Y ∧ noDups(U)
                    ∧ setOf(U) !! h.Keys
                    ∧ extVal(h,U,Z).Values ≤ B.Dom
                    ∧ models(B,extVal(h,U,Z),existSq(W,phii));
        extValDomRange_Lemma(h, U, Z);
        assert extVal(h,U,Z).Keys == j'.V;
        extValallMaps_Lemma(h, U, Z, B);
        assert extVal(h,U,Z) in allMaps(j'.V, B.Dom);
        assert valuationModel(extVal(h,U,Z),j',phi,B);
        //assert extVal(h,U,Z) in j'.F; // by hypothesis (H2)
        //assert h.Keys == j.V;
        projectOfExtVal_Lemma(h, U, Z);
        assert projectVal(extVal(h,U,Z),j.V) == h;
        //assert j.F == ( set f | f in j'.F • projectVal(f,j.V) );
                      // by hypothesis (H1)
        //assert h in j.F;
        }
```

Firstly, we define the variable `phii` which represents the subformula $\phi(i)$ of index i of the parameter formula ϕ (denoted in code by `phi`). Then, we define the three sets of variables `W`,`Y` and `X` occurring in $\phi(i)$ and in the judgements `j` and `j'`. Next, we proof that any valuation h such that $B, h \models \exists X(\phi(i))$ belongs to `j.F`. This is the meaning of the $\forall$-`ensures` in the code, whose proof is inside the curly brackets that completes the proof. This proof calls five auxiliary lemmas, but it is easy to follow because the assertions after each lemma call explain what they add to prove. By the hypothesis, we have that $B, h \models \exists X(\phi(i))$ where $X = Y + W$, from here, the auxiliary lemma `existSq_Sum_Lemma` ensures that $B, h \models \exists Y \exists W(\phi(i))$. Then, by auxiliary lemma `existSqSem_Lemma`, we basically prove that $B, h[Y \mapsto Z] \models \exists W(\phi(i))$ for some values set of values Z in B. In the code, `U` is a sequence representing the set Y with no repetitions and disjoint with `h.Keys` and `extVal(h,U,Z)` is the Dafny code for $h[Y := Z]$. After the calls to lemmas `extValDomRange_Lemma` and `extValDomRange_Lemma` we prove that hypothesis `(H2)` can be applied to the mapping $h[Y \mapsto Z]$ so that $h[Y \mapsto Z] \in j'.F$ holds. Therefore, its projection $h[Y \mapsto Z] \upharpoonright j.V$, which is proved to coincide with the mapping h (using the auxiliary lemma `projectOfExtVal_Lemma`), should belong to $j.V$. The latter is due to hypothesis `H1`, since by definition of projection $j.F$ is the set of all projection on $j.V$ of all valuations in $j'.F$. Therefore, $h \in j.F$ is proved. The other three inductive cases –for derivability using (join), ($\forall$-elimination), and (upward flow)– follows the same lines of the case for (projection) and their code is above omitted. Next, the `soundness_Theorem`, which states the soundness of the proof system PS, can be easily proved by calling the previous `models_Lemma`.

```
lemma soundness_Theorem<T> (phi:Formula,B:Structure<T>)
requires wfQCSP_Instance(phi,B)
requires is_derivable(J([],{},{}),phi,B)
ensures ¬models(B,map[],phi)
{
var cj := J([],{},{});
models_Lemma(cj,phi,B);
assert ¬valuationModel(map[],cj,phi,B);
}
```

Since the empty judgement is derivable, by `models_Lemma`, every valuation that is a model of `phi` belongs to the empty set of valuations. Therefore, every possible valuation with empty domain is not a valuation model of `phi`. Since the empty function `map[]` is the only valuation in the set of valuations with empty domain, then `(B,map[])` cannot models `phi`.

Completeness, i.e. the backward direction of Theorem 1, is encoded in the following `completeness_Theorem` which is proved with the help of the following auxiliary lemma:

Lemma 2 *Let $(\phi, \mathbf{B})$ be a QCSP instance. Let I_ϕ be the index set of ϕ. For each $i \in I_\phi$, let F be the set of all valuations such that $\mathbf{B}, h \models \phi(i)$. Then, the judgement $(i, \mathsf{freeVar}(\phi(i)), F)$ is derivable.*

We provide a constructive proof of Lemma 2 that associates a judgement to each index `i` of the formula ϕ. We call it the *canonical judgement*. It is recursively defined by the following function:

```
function canonical_judgement<T> (i:seq<int>,phi:Formula,
                                 B:Structure<T>): (cj:Judgement<T>)
  requires wfQCSP_Instance(phi,B)
  requires i in setOfIndex(phi)
  ensures cj.i = i
  ensures cj.V = freeVar(FoI(i,phi,B.Sig))
  ensures wfJudgement(cj,phi,B)
  decreases FoI(i,phi,B.Sig)
{
var phii := FoI(i,phi,B.Sig);
indexSubformula_Lemma(i,phi,B.Sig);
match phii
case Atom(R,par) => var F := (set f:Valuation<T> |
                                f in allMaps(setOf(par),B.Dom)
                                ∧ HOmap(f,par) in B.I[R]);
                    J(i,setOf(par),F)
case And(phi0,phi1) => var j0' := canonical_judgement(i+[0],phi,B);
                       var j1' := canonical_judgement(i+[1],phi,B);
                       var j0 := J(i,j0'.V,j0'.F);
                       var j1 := J(i,j1'.V,j1'.F);
                       join(j0,j1,phi,B)
case Forall(x,phik) => var j0 := canonical_judgement(i+[0],phi,B);
                       if x in j0.V then dualProjection(x,j0,phi,B)
                                    else J(i,j0.V,j0.F)
case Exists(x,phik) => var j0:= canonical_judgement (i+[0],phi,B);
                       var jp := projection(j0,j0.V-{x},phi,B);
                       if x in j0.V then J(i,jp.V,jp.F)
                                    else J(i,j0.V,j0.F)
}
```

Note that we name by `cj` the result produced by the function `canonical_judgement`. Therefore, the name `cj` is used in the specification of the function as the shorter name of `canonical_judgement(i, phi, B)`. The well-foundedness of this function is given by the decreasing expression `FoI(i,phi,B.Sig)`, which represents the subformula of `phi` whose index is `i` (or ($\phi(i)$)). The call to `indexSubformula_Lemma` ensures that the indexes of the subformulas of `phi,phi1,phik`, in the succeeding recursive definition given by a `match` statement, are correct. In that recursive definition, we use three auxiliary functions `join`, `dualProjection` and `projection` whose definitions has been ensured to satisfy the respective predicates `is_join`, `is_dualProjection` and `is_projection` (see Section 4). Consequently, Lemma 2 is encoded in the following Dafny lemma

```
lemma canonical_judgement_Lemma<T>(i:seq<int>,phi:Formula,
                                    B:Structure<T>,cj:Judgement<T>)
  requires wfQCSP_Instance(phi,B) ∧ i in setOfIndex(phi)
  requires cj = canonical_judgement(i,phi,B)
  ensures cj.F = setOfValmodels(FoI(i,phi,B.Sig),B)
  ensures is_derivable(cj,phi,B)
  decreases FoI(i,phi,B.Sig)
⊞{...}
```

that uses the following function

```
function setOfValModels<T> (phi:Formula,B:Structure<T>)
                                              : set<Valuation<T>>
requires wfStructure(B) ∧ wfFormula(B.Sig,phi)
{
(set f:Valuation<T> | f in allMaps(freeVar(phi),B.Dom)
                       ∧ models(B,f,phi))
}
```

for representing the set of all valuations that are models of a given formula in a given structure. The lemma `canonical_judgement_Lemma` is proved by structural induction on `phi` (we do not show here the proof), with the help of auxiliary lemmas given inductive properties of the definition of `setOfValModels` for the three different types of composed formulas (`And`, `Forall` and `Exists`) in terms of their component subformula(s). Next, the `completeness_Theorem`, calling `canonical_judgement_Lemma`, proves that whenever the sentence `phi` is not satisfied by the structure `B`, then the empty judgement of index `[]` is derivable, indeed it is the cannonical judgement for `([], phi, B,)`.

```
lemma completeness_Theorem<T> (phi:Formula,B:Structure<T>)
  requires wfQCSP_Instance(phi,B)
  requires ¬models(B,map[],phi)
  ensures is_derivable(J([],{},{}),phi,B)
{
var cj :| cj = canonical_judgement([],phi,B);
canonical_judgement_Lemma([],phi,B);
   //assert cj.V = {};
   //assert cj.F = (set f:Valuation<T> | f in allMaps({}, B.Dom)
   //                            ∧ models(B,f,phi)) = {};
  }
}
```

The completeness proof proceeds by calling the `canonical_judgement_Lemma` with canonical judgement associated to the root index `[]`, Dafny infers that this judgement `cj` is derivable. Moreover, `cj.V` must be empty and `cj.F` is the set of all valuations with empty domain that are valuation models of `phi` (paired with `B`) must be empty.

6 Modular Structure

In previous sections, we described the essentials of our formalization and the proofs of the main meta-logical properties: soundness and completeness. However, many technical details and auxiliary properties are proved for that. In our opinion, a modular structure with explicit declarations of the definitions and lemmas that are exported from one module and imported in other module, is essential for refactoring and reuse a large formalization. Moreover, in our experience, modularity demonstrates to be helpful during the development phase. In this section, we give an idea of the whole encoding by describing how it is structured using modules.

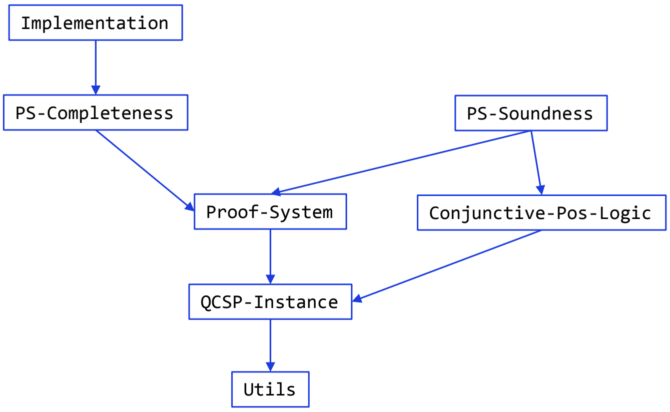

Fig. 2 Module dependencies (w.r.t. `include` clauses)

Dafny provides *modules* (keyword `module`) to group together related entities (such as datatypes, lemmas, functions, predicates, methods, etc.), as well as to control the scope of declarations. and clauses `include` to include one module in other. In the head of modules, Dafny allows clauses `import` and `export` and different qualifiers. In the case of `import` the qualifier `opened` allows to use the name of the imported units without the additional prefix of the imported module name. By declaring an export set, a module makes available a subset of its declarations to the module's importers. In the case of `export`, Dafny supports multiple export sets per module and also allows to name the different exported list. In addition, the qualifiers `provides` and `reveals` allows us to export respectively the *specification-part* or also the *body-part* of the exported unit. In other words, each export set indicates the translucency of its exported declarations. For a function, the specification-part includes the function's parameters/results type signature as well as the function's specification, whereas the body-part includes its definition. In Dafny, the verifier always reasons about calls to lemmas (also to methods) in terms of their specifications, never in terms of their bodies. Thus, there would be no difference between providing and revealing a method or lemma in an export set. For that reason methods and lemmas are disallowed to be mentioned in reveals clauses. Therefore, exported lemmas are always provided, but not revealed. An export set has to be *self consistent*. This means that everything mentioned in the exported declarations must make sense separately. In particular, this means that every symbol that is mentioned in the portions (specification-part or specification-part plus body-part) that are exported must also be part of the export set. The interested reader can find motivations and explanations about the design of the module system of Dafny in [29].

Our formalization is structured in seven modules whose dependencies are described in Figure 2, where the arrows represent the clauses `include` in each module. The module at the tail of each arrow includes the one at the head of the arrow. The proof system formalization and the proofs of its soundness and completeness consists of six modules: `Utils`, `QCSP-instance`, `Proof-System`, `Conjunctive-Pos-Logic`, `PS-Soundness` and `PS-Completeness`. In addition, the module `Implementation` contains some additional (verified) methods necessary for implementing the model checker as a web application. More details on the latter are given in Section 7.

Module `Utils` contains a few auxiliary concepts and properties on sets, sequences and maps that are of general utility. Here we show part of it:

```
module Utils {
export reveals setOf, noDups
       provides allMaps,allMaps_Correct_Lemma,ExtMap_Lemma,
                ProyectMap_Lemma

function setOf<T>(s:seq<T>): set<T>
{ set x | x in s }

predicate noDups<T(==)>(U:seq<T>)
{ ∀ i,j • 0 ≤ i < j < |U| ⟹ U[i] ≠ U[j] }

function allMaps<A,B>(keys: set<A>, values: set<B>): set<map<A,B>>
ensures ∀ m • m in allMaps(keys, values)
              ⟹ m.Keys = keys ∧ m.Values ⊆ values
ensures keys = {} ⟹ allMaps(keys,values) = {map[]}
⊞{...}

...//one (unrevealed) auxiliary function

...//lemmas allMaps_Correct_Lemma, ExtMap_Lemma and ProyectMap_Lemma
}
```

The export list of module `Utils` reveals a function `setOf` for the set of elements of a sequence, a predicate `noDups` for deciding whether a given sequence has duplicates, which has not ensures clauses. However, it provides (but does not reveal) the function `allMaps` (see Section 4), hence importer modules know its two ensure clauses, but not its body. Module `Utils` also provides a lemma `allMaps_Correct_Lemma`, which complements the first ensures clause of `allMaps` with the backward implication, along with other two lemmas on operations over sets of maps.

In Sections 4 and 5 we explained the most revelant componentes of the four modules `QCSP-Instance`, `Proof-System`, `PS-Soundness` and `PS-Completeness`. In what follows, we explain the role of the module `Conjunctive-Pos-Logic` in our formalization. For that, we first give more details about the module `QCSP-Instance`. This module contains 19 lemmas proving basic properties of the operations on valuations, and also properties on the relation between these operations and the predicate `models`. Some of this units are auxiliary in the module, to prove the lemmas that are provided to other modules. Next, we show a partial view of the module `QCSP-Instance` placing emphasis on the relevant elements in the export list to the module `Conjunctive-Pos-Logic`.

```
module QCSP_Instance {
import opened Utils
export Lemmas_for_Conj_Pos_Logic
       reveals ..., extVal, ...
       provides ..., extValDomRange_Lemma, extValOrder_Lemma,
                NoFreeVarInExists_Lemma, Exists_Commutes_Lemma
   ... // export lists for other modules

function extVal<T>(f:Valuation<T>,W:seq<Name>,S:seq<T>)
                                                     : Valuation<T>
requires |W| = |S| ∧ noDups(W)
decreases W
{ if W = [] then f else extVal(f[W[0]:=S[0]],W[1..],S[1..]) }

... // other function and predicate definitions

lemma extValDomRange_Lemma<T>(f:Valuation<T>,W:seq<Name>,S:seq<T>)
requires |W| = |S| ∧ noDups(W)
ensures extVal(f,W,S).Keys = setOf(W) + f.Keys
ensures extVal(f, W, S).Values ⊆ f.Values + setOf(S)
decreases W
⊞{...}

lemma extValOrder_Lemma<T>(k:int,U:seq<Name>,S:seq<T>,f:Valuation<T>)
requires 0 ≤ k < |U| = |S| ∧ noDups(U)
ensures extVal(f, U, S)
          = extVal(f[U[k]:=S[k]], U[..k]+U[k+1..], S[..k]+S[k+1..])
⊞{...}

... // other 15 lemmas

lemma NoFreeVarInExists_Lemma<T>(B:Structure,f:Valuation<T>,
                                  x:Name,beta:Formula)
requires wfStructure(B) ∧ wfFormula(B.Sig,beta) ∧ f.Values ⊆ B.Dom
requires x ∉ freeVar(beta)
ensures models(B,f,beta) ⟺ models(B,f,Exists(x,beta))
⊞{...}

lemma Exists_Commutes_Lemma<T>(x:Name, y:Name, alpha:Formula,
                                f:Valuation<T>, B:Structure<T>)
requires wfStructure(B) ∧ wfFormula(B.Sig, alpha)
requires f.Values ⊆ B.Dom
requires models(B, f, Exists(x, Exists(y, alpha)))
ensures models(B, f, Exists(y, Exists(x, alpha)))
⊞{...}
```

Dotted lines in the export list substitute the elements that are necessary for self consistency, but are not relevant for the present discussion. In other words, 4 of the 19 lemmas proved in `QCSP-Instance` are basic for proving the 13 lemmas in module `Conjunctive-Pos-Logic`. The objective of the latter module is to provide the properties of Conjunctive Positive Logic that we need to prove soundness. Indeed, all them are properties about the models of formulas of the form $\exists x_1 \dots \exists x_n \phi$, which is written in Dafny as `existSq(W,phi)` where `W` is the sequence $[x_1 \dots x_n]$.

The function `existSq` is defined and exported by module `Conjunctive-Pos-Logic`, see Figure 3.

```
module Conjunctive_Pos_Logic{
...// import opened clauses

export Lemmas_for_PS_Soundness
  reveals existSq
  provides existSq_ExtVal_Lemma, existSq_Project_Lemma,
             existSq_Sum_Lemma, existSq_And_Lemma,
             existSq_Forall_Lemma, existSq_Exists_Lemma,
             existSqSem_Lemma

function existSq(X:set<Name>, alpha:Formula): Formula
ensures freeVar(existSq(X,alpha)) = freeVar(alpha)-X
ensures ∀ S • wfFormula(S,alpha) ⟹ wfFormula(S,existSq(X,alpha))
{
if |X| = 0 then alpha else var x :| x in X;
                              Exists(x,existSq(X-{x},alpha))
}

lemma existSq_And_Lemma<T>(B:Structure<T>,f:Valuation<T>,
                              W:set<Name>,phi:Formula)
requires wfStructure(B) ∧ wfFormula(B.Sig,phi)
requires f.Values ⊆ B.Dom
requires phi.And?
requires models(B,f,existSq(W,phi))
ensures wfFormula(B.Sig,existSq(W ∩ freeVar(phi.0),phi.0))
ensures wfFormula(B.Sig,existSq(W ∩ freeVar(phi.1),phi.1))
ensures models(B,f,existSq(W ∩ freeVar(phi.0), phi.0))
ensures models(B,f,existSq(W ∩ freeVar(phi.1), phi.1))
decreases W
⊞{...}

... // other 12 lemmas
}
```

Fig. 3 One of the lemmas exported from `Conjunctive-Pos-Logic` to prove the soundness of proof system *PS*.

The specification-part of lemma `existSq_Distr_And_Lemma` is shown in Figure 3 as an example of the kind of properties about models of `existSq`-formulas that module `Conjunctive-Pos-Logic` provides to module `PS-Soundness`. Basically, it proves that

if $B, f \models \exists x_1 \ldots \exists x_n(\phi_0 \ \wedge \ \phi_1)$,
then $B, f \models \exists y_1 \ldots \exists y_m(\phi_0)$ and $B, f \models \exists z_1 \ldots \exists z_k(\phi_1)$

where $\{y_1, \ldots, y_m\}$ is the set of all variables in $\{x_1, \ldots, x_n\}$ that occur free in ϕ_0 and $\{z_1, \ldots, z_k\}$ is the set of all variables in $\{x_1, \ldots, x_n\}$ that occur free in ϕ_1. All the lemmas exported (provided) by the module `Conjunctive-Pos-Logic` are related to (semantic) models of CPL. In particular, the seven lemmas exported by module `Conjunctive-Pos-Logic` to be imported by the module `PS-Soundness`, (see export list `Lemmas_for_PS_Soundness` in Figure 3) assist in the task of proving the lemma `models_Lemma` that is crucial in the soundness proof, as explained in Section 5. Since `models_Lemma` (or Lemma 1) refers to an existentially closed formula, the metalogical properties in module `Conjunctive-Pos-Logic` are related to these existentially closed formulas.

7 Implementation

On the basis of our formalization of proof system PS, we have implemented a verified model checker for Conjunctive Positive Logic. The interface of our model checker asks the user to successively provide the different components of a QCSP-instance $(\mathbf{B}, \phi)$, and when completed it returns the result of whether $\mathbf{B} \models \phi$. In this section, we describe our conversion process to generate code and integrate it into the web application. We report on the challenges that arise along this process.

To obtain code from the verified proof system, we basically convert the functions (and predicates) that would take part in the implementation of the web application into methods. By default, Dafny functions (and predicates) are ghost (non-executable), and cannot be called from non-ghost code. Predicates receive the same treatment as functions, they really are boolean functions. To make a function non-ghost, Dafny gives the option, when feasible, to replace the keyword `function` with the two keywords `function method`. When a function `f` defined by an expression `E` is turned to non-ghost, every function called in the expression `E` should be turned to non-ghost. Not every expression can be changed from ghost to non-ghost, because not every ghost expression is compilable into real code. As a typical example, consider any expression of the form `∀ i: nat • P(i)` that is body/definition of a `predicate Q`. If property `P` does not bound the possible values of `i` in some way that enables Dafny's heuristics to get a finite set, then the change to `predicate method Q` raises an error in Dafny that complains: "a quantifier in a non-ghost context is allowed only whenever a bounded set of values for its variables (`i`, in this case) can be computed". In this case, the required function (or predicate) should be implemented by a method whose requires-ensures specification (a.k.a. contract) states that it computes the original function.

The web application checks the well-formedness of the QCSP-instance given by the user, and then compute the canonical judgement to answer "no" if it is empty and "yes" otherwise. Henceforth, at a first glance, we have to convert into non-ghost the predicate `wfQCSP_Instance` and the function `canonical_judgement`. As a consequence, all ghost code used in each of these three units has to be also transformed into real code, and the same applies to the ones called from the just transformed into non-ghost. We made this until Dafny has not more complains telling us that"function calls are allowed only in specification context (consider declaring the function as function method)". Dafny marks the affected calls and shows that messages as hover text, which is a valuable help. The predicate `wfQCSP_Instance` is easily turned non-ghost by simply adding the keyword `method` in other two predicates and functions definitions. The transformation of function `canonical_judgement` requires that eight different functions must be also non-ghost. Five of them are solved by simply adding the keyword `method`. One of the other three functions, which is `allMaps`, does not satisfy the required conditions for that easy conversion into code, whereas the other two (`join` and `dualProjection`) call `allMaps`. Indeed, the function `allMaps` makes use of the Hilbert epsilon operator ([32]) declaring `var a :| a in s`, where `s` is a set, raises the error "to be compilable the value of a let-such-that-expression must be uniquely determined". To fix this problem, we developed the module `Implementation` in which we provide a method `compute_f` for each of the four functions `canonical_judgement`, `allMaps`, `join` and `dualProjection` as `f`, and verify the equivalence of each method with the original function. Actually the contract of the methods `compute_f` specify that it conputes the function `f`. For example, the contracts of `compute_canonical_judgement` and `compute_allMaps` are:

```
method compute_canonical_judgement<T>(i:seq<int>,phi:Formula,
                                      B:Structure<T>)
                                      returns (cj:Judgement<T>)
requires wfQCSP_Instance(phi,B)
requires i in setOfIndex(phi)
ensures cj = canonical_judgement(i,phi,B)
```

```
method compute_allMaps<T(==)>(keys:set<Name>,values:set<T>)
                                  returns (am:set<map<Name,T>>)
requires values ≠ {}
ensures am = allMaps(keys, values)
```

After that, by compiling our formalization, Dafny automatically generates a library of methods in .NET code (i.e. C#, Visual Basic, and F#). Since .NET does not have a standard format for inductive datatypes, the data format used by the Dafny compiler may not agree with the data formats used by other .NET languages. Therefore, the use of our verified encoding from C# has required some data conversions. The compilation process transforms `datatype`s, such as `Structure`, `Judgement` and `Formula`, into C# classes. For each constructor `C` of a datatype `D` a class is created named as `D_C`. All these classes extend a generic abstract one called `Base_D`. In addition, there is a class `D` that has a single constructor with a parameter of type `Base_D`. To illustrate this, the `datatype Formula` has four constructors in Dafny specification: `Atom, And, Forall` and `Exists`. Dafny generates the classes `Formula, Base_Formula` (abstract), `Formula_Atom, Formula_And, Formula_Exists` and `Formula_Forall`. Auxiliary functions are automatically generated to help developers to handle with the classes. For example, a function `is_C` is generated for each constructor `C`. With the previous classes and auxiliary functions we can instantiate C# objects. These can be used as input in methods that require them.

The fact that our formalization has already been verified guarantees that every call that satisfies the precondition complies with the postcondition. As a consequence, our web application checks the `requires` clauses before calling the methods. In other words, the only method called by the web application is `compute_canonical-judgement` whose preconditions are:

```
requires wfQCSP_Instance(phi,B)
requires i in setOfIndex(phi)
```

Hence, before the call `compute_canonical-judgement([],pbi,B)`, we only check that `wfQCSP_Instance(phi,B)`, because `[] in setOfIndex(phi)` is trivial for any `phi`.

8 Experience

Our formalization and implementation has been developed in the Dafny IDE ([31]) which lends itself to increase user productivity. The main features of the Dafny IDE are well described in [31]. Our development experience can be termed as highly positive, mainly because interaction with the tool is easy and it provides good support and helpful information for verification failures, in an agile and fast way. In addition, Dafny supports a number of proof features traditionally found in only interactive proof assistants like Coq or Isabelle/HOL. The task is interesting from the point of view of given a detailed formalization and for debugging the proofs which were previously written with pen and paper in a more imprecise form. Automated proofs often require proving some essential properties that are usually assumed (in the concerned area) without any proof. This is especially the case of many of the lemmas in the module `Conjunctive-Pos-Logic` which mainly contain logical equivalences that are usually assumed. Moreover, it is feasible that the process of proving some of these properties raised some issue non-properly defined in the formalization. Actually, this was our experience, as we explain in the next paragraph.

There is no doubt that formal verification is useful and important in software development. Details can be subtle and formal verification helps in detecting subtle details that otherwise remain in hiding. The more noteworthy are assumptions that the programmer assume, but she (or he) does not make explicit in the specification. Along the development of our proof many subtle details has been fixed. For example, we forgot to specify, as part of the predicate `wfStructure`, that the domain of the given structure must be warranted to be non-empty. We realized that when we were not able to proof lemma `existSqSem_Lemma` in module `Conjunctive-Pos-Logic`. The interested reader could commented the first line in the body of predicate `wfStructure` to check that the proof of lemma `existSqSem_Lemma` is not verified.

Another worthy mistake we made was when we defined the canonical judgement for the universal and existential formulas without taking into account the case when the quantified variable is not in the variables of the judgement. Hence, when we initially tried to prove `canonical_judgement_Lemma`, the postcondition:

```
ensures cj.F == setOfValmodels(FoI(i,phi,B.Sig),B)
```

couldn't be proved, leading us to see where we were missing.

Among the many interesting lessons learned, we would like to report on the details of the definition of `function allMaps` in module `Utils`. For that, we use the function

```
function choose<A>(s: set<A>): A
requires s ≠ {}
{
var a :| a in s; a
}
```

to encapsulate into this function the application of the Hilbert epsilon operator `:|`. Otherwise, if we used the operator `:|`directly in two places: the definition of `function allMaps` and the proof of `lemma allMaps_Correct_Lemma`, then each place would choose a different element, which makes the proof of the lemma much harder than putting the expression into a function, because a function produces a unique result for a given argument.

The Dafny formalization, which consists of 1963 lines (including white and commented lines), is structured in seven modules. We take advantage of the import/export mechanism of Dafny for organizing the dependencies between the components. Moreover, in Dafny, the exported components can be provided or revealed, which enables to export just the specification, or also the body, of functions and methods.

module	lines	datatypes	functions	lemmas	methods	proof obl.	secs
`Utils`	65	0	4	3	0	39	2.90
`QCSP-Instance`	358	2	8	19	8	697	16.99
`Proof-System`	218	1	7	4	1	469	14.22
`Conjunctive-Pos-Logic`	363	0	2	13	0	1,042	46.34
`PS-Soundness`	288	0	1	8	0	1,004	32.22
`PS-Completeness`	231	0	4	5	1	698	17.84
`Implementation`	204	0	0	4	7	256	10.58
TOTAL	1727	3	26	56	17	4,205	141.09

Table 1 Some figures on lines, different Dafny units (datatypes, functions, ...), generated proof obligations, and seconds required for verification, breakdown per module.

In Table 1 we summarize the size of the modules, in terms of the total (non-comment, non-blank) lines of code, the number of datatypes, the number of functions (including predicates), proved lemmas, and methods. The function/predicate methods are counted as methods. The right-most two column respectively reports the number of proof obligations (i.e. queries discharged to Z3) and the seconds required to verify each module by Dafny 2.3.0 for Windows(x64) running by a processor i5-7500 CPU at 2.60GHz 3.40GHz with 16 GB of RAM.

The proof of lemma `existSq_Distr_And_Lemma` takes about 17,45 seconds, which is the most costly proof, for solving 152 proof obligations. However, the largest set of proof obligations, which consists of 423, is generated by `models_Lemma_h` is proved in 0,42 seconds. To give some global figures about the proof obligations (PO) generated by lemmas, from the 51 lemmas, there are 18 lemmas that generate at most 20 PO, 16 lemmas that generate from 21 to 60 proof obligations, 21 lemmas that generates from 61 to 250, and only one lemma that produces more than 250 which is the above mentioned `models_Lemma_h`. Just below it, the lemma `canonical_judgement_Lemma` produces the number of PO closer to 250, exactly 218, and it is verified in 9,06 seconds.

The amount of effort required to develop the whole system (seven modules of formalization and the web application) is about 250 person-hours.

9 Compliance with Ethical Standards

Funding: This research has been supported by the European Union (FEDER funds) under grant TIN2017-86727-C2-2-R, and by the University of the Basque Country under Project LoRea GIU18-182.
Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. Abuin, A., Chen, H., Hermo, M., Lucio, P.: Towards the automatic verification of QCSP tractability results. In: Proceedings of the XVII Jornadas sobre Programación y Lenguajes (PROLE 2017) (2017). URL `http://hdl.handle.net/11705/PROLE/2017/017`
2. Backhouse, R. (ed.): The calculational method, *Information Processing Letters*, vol. 53. Elsevier (1995). DOI 10.1016/0020-0190(94)00212-H
3. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Formal Methods in System Design **41**(1), 45–65 (2012). DOI 10.1007/s10703-012-0152-6
4. Balabanov, V., Widl, M., Jiang, J.H.R.: QBF resolution systems and their proof complexities. In: C. Sinz, U. Egly (eds.) Theory and Applications of Satisfiability Testing – SAT 2014, pp. 154–169. Springer International Publishing, Cham (2014)
5. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, pp. 4786–4790 (2017). DOI 10.24963/ijcai.2017/667. URL `https://doi.org/10.24963/ijcai.2017/667`
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's Verify This with Why3. Software Tools for Technology Transfer (STTT) **17**(6), 709–727 (2015). DOI 10.1007/s10009-014-0314-5. URL `https://hal.inria.fr/hal-00967132`
7. Bordeaux, L., Monfroy, E.: Beyond NP: Arc-consistency for quantified constraints. In: P. Van Hentenryck (ed.) Principles and Practice of Constraint Programming - CP 2002, pp. 371–386. Springer Berlin Heidelberg (2002). DOI 10.1007/3-540-46135-3_25
8. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda — a functional language with dependent types. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs'09, pp. 73–78. Springer-Verlag (2009). DOI 10.1007/978-3-642-03359-9_6
9. Buning, H., Karpiński, M., Flogel, A.: Resolution for quantified boolean formulas. Information and Computation **117**(1), 12 – 18 (1995). DOI 10.1006/inco.1995.1025
10. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An algorithm to evaluate quantified boolean formulae and its experimental evaluation. Journal of Automated Reasoning **28**(2), 101–142 (2002). DOI 10.1023/A:1015019416843
11. Chen, H.: A rendezvous of logic, complexity, and algebra. ACM Computing Surveys **42**(1), 2:1–2:32 (2009). DOI 10.1145/1592451.1592453
12. Chen, H.: A rendezvous of logic, complexity, and algebra. ACM Comput. Surv. **42**(1), 2:1–2:32 (2009). DOI 10.1145/1592451.1592453
13. Chen, H.: Beyond Q-resolution and prenex form: A proof system for quantified constraint satisfaction. Logical Methods in Computer Science **10**(4) (2014). DOI 10.2168/LMCS-10(4:14)2014
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: D. Kozen (ed.) Logics of Programs, pp. 52–71. Springer Berlin Heidelberg (1982). DOI 10.1007/BFb0025774
15. Clochard, M., Filliâtre, J.C., Marché, C., Paskevich, A.: Formalizing Semantics with an Automatic Program Verifier, pp. 37–51. Springer International Publishing (2014). DOI 10.1007/978-3-319-12154-3_3
16. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, pp. 23–42. Springer (2009). DOI 10.1007/978-3-642-03359-9_2
17. Creignou, N., Khanna, S., Sudan, M.: Complexity classifications of boolean constraint satisfaction problems, *SIAM Monographs on Discrete Mathematics and Applications,*, vol. 7. Society for Industrial and Applied Mathematics (2001). DOI 10.1137/1.9780898718546
18. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc. (2003). DOI 10.1016/B978-1-55860-890-0.X5000-2
19. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified boolean formulas. In: H.A. Kautz, B.W. Porter (eds.) Proceedings of the 17th National Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence, pp. 417–422. AAAI Press / The MIT Press (2000). URL `http://www.aaai.org/Library/AAAI/2000/aaai00-064.php`
20. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable ltl model checker. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification, pp. 463–478. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

21. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: M. Felleisen, P. Gardner (eds.) Programming Languages and Systems — 22nd European Symposium on Programming, ESOP 2013, *Lecture Notes in Computer Science*, vol. 7792, pp. 125–128. Springer (2013). DOI 10.1007/978-3-642-37036-6_8
22. Gelder, A.V.: Contributions to the theory of practical quantified boolean formula solving. In: M. Milano (ed.) Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7514, pp. 647–663. Springer (2012). DOI 10.1007/978-3-642-33558-7_47
23. Gent, I.P., Nightingale, P., Rowley, A., Stergiou, K.: Solving quantified constraint satisfaction problems. Artificial Intelligence **172**(6), 738 – 771 (2008). DOI 10.1016/j.artint.2007.11.003
24. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for quantified boolean logic satisfiability. Artificial Intelligence **145**(1-2), 99–120 (2003). DOI 10.1016/S0004-3702(02)00373-9
25. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. Journal of The ACM pp. 552–552 (2007)
26. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: M. Bobaru, K. Havelund, G. Holzmann, R. Joshi (eds.) NASA Formal Methods, pp. 41–55. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-20398-5_4
27. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-aided reasoning : an approach. Advances in formal methods. Kluwer Academic Publishers (2000). DOI 10.1007/978-1-4615-4449-4
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: E.M. Clarke, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, *Lecture Notes in Computer Science*, vol. 6355, pp. 348–370. Springer (2010). DOI 10.1007/978-3-642-17511-4_20
29. Leino, K.R.M., Matichuk, D.: Modular verification scopes via export sets and translucent exports. In: Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday, pp. 185–202 (2018). DOI 10.1007/978-3-319-98047-8_12
30. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: E. Cohen, A. Rybalchenko (eds.) Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 8164, pp. 170–190. Springer (2014). DOI 10.1007/978-3-642-54108-7_9
31. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: C. Dubois, D. Giannakopoulou, D. Méry (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, *Electronic Proceedings in Theoretical Computer Science*, vol. 149, pp. 3–15. Open Publishing Association (2014). DOI 10.4204/eptcs.149.2
32. Leino, R.: Compiling hilbert's epsilon operator. In: A. Fehnker, A. McIver, G. Sutcliffe, A. Voronkov (eds.) LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, *EPiC Series in Computing*, vol. 35, pp. 106–118. EasyChair (2015). DOI 10.29007/rkxm. URL `https://easychair.org/publications/paper/dM`
33. Leino, R.: Well-founded functions and extreme predicates in Dafny: A tutorial. In: B. Konev, S. Schulz, L. Simon (eds.) IWIL-2015. 11th International Workshop on the Implementation of Logics, *EPiC Series in Computing*, vol. 40, pp. 52–66. EasyChair (2016). DOI 10.29007/v2m3
34. Mamoulis, N., Stergiou, K.: Algorithms for quantified constraint satisfaction problems. In: Proceedings of CP'04, *LNCS*, vol. 3258, pp. 752–756. Springer (2004)
35. Martin, B.: Dichotomies and duality in first-order model checking problems. CoRR (2006). URL `http://arxiv.org/abs/cs/0609022`
36. Martin, B.: First-order model checking problems parameterized by the model. In: A. Beckmann, C. Dimitracopoulos, B. Löwe (eds.) Logic and Theory of Algorithms, pp. 417–427. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-69407-6_45
37. Martin, B.: Quantified constraints in twenty seventeen. In: A. Krokhin, S. Zivny (eds.) The Constraint Satisfaction Problem: Complexity and Approximability, *Dagstuhl Follow-Ups*, vol. 7, pp. 327–346. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017). DOI 10.4230/DFU.Vol7.15301.327

38. Marx, D.: Tractable hypergraph properties for constraint satisfaction and conjunctive queries. Journal of The ACM **60**(6) (2013). DOI 10.1145/2535926
39. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
40. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: M. Dezani-Ciancaglini, U. Montanari (eds.) International Symposium on Programming, pp. 337–351. Springer Berlin Heidelberg (1982). DOI 10.1007/3-540-11494-7_22
41. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: Qed at large: A survey of engineering of formally verified software. Foundations and Trends® in Programming Languages **5**(2-3), 102–281 (2019). DOI 10.1561/2500000045. URL `http://dx.doi.org/10.1561/2500000045`
42. Rintanen, J.: Constructing conditional plans by a theorem-prover. Journal of Artificial Intelligence Research **10**(1), 323–352 (1999). URL `http://dl.acm.org/citation.cfm?id=1622859.1622870`
43. Schaefer, T.J.: The complexity of satisfiability problems. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78, pp. 216–226. ACM, New York, NY, USA (1978). DOI 10.1145/800133.804350. URL `http://doi.acm.org/10.1145/800133.804350`
44. Schlichtkrull, A.: Formalization of the resolution calculus for first-order logic. Journal of Automated Reasoning **61**(1–4), 455–484 (2018). DOI 10.1007/s10817-017-9447-z
45. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, p. 152–165. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3293880.3294100. URL `https://doi.org/10.1145/3293880.3294100`
46. Stockmeyer, L.: The Complexity of Decision Problems in Automata Theory and Logic. MAC TR. Massachusetts Institute of Technology, Project MAC (1974). URL `https://books.google.es/books?id=zFbQMQAACAAJ`
47. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. Journal of Functional Programming **23**(4), 402–451 (2013). DOI 10.1017/S0956796813000142
48. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285–309 (1955). URL `https://projecteuclid.org:443/euclid.pjm/1103044538`
49. The Coq Development Team: The logical project, INRIA. The Coq proof assistant (Version 8.10.0, 2019). URL `https://coq.inria.fr`
50. Williams, R.: Algorithms for quantified boolean formulas. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, pp. 299–307. Society for Industrial and Applied Mathematics (2002). URL `http://dl.acm.org/citation.cfm?id=545381.545421`
51. Xavier, B., Olarte, C., Reis, G., Nigam, V.: Mechanizing focused linear logic in Coq. In: The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017), *Electronic Notes in Theoretical Computer Science*, vol. 338, pp. 219 – 236 (2018). DOI 10.1016/j.entcs.2018.10.014
52. Zhang, L., Malik, S.: Conflict driven learning in a quantified boolean satisfiability solver. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, ICCAD '02, p. 442–449 (2002). DOI 10.1145/774572.774637