SN Computer Science (2021) 2:344
https://doi.org/10.1007/s42979-020-00417-3

ORIGINAL RESEARCH

Verified Model Checking for Conjunctive Positive Logic

Alex Abuin[2] · Unai Diaz de Cerio[2] · Montserrat Hermo[1] · Paqui Lucio[1]

Received: 8 June 2020 / Accepted: 2 December 2020 / Published online: 19 June 2021

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's automatic code generation, from the completeness 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 decides 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

Introduction

Model checking [1,2] 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, a.k.a. first-order model checking [3,4], is extensively studied in complexity theory and it has interesting applications in finite model finding and database theory. In general, the model checking problem is intractable. To be precise, the model checking for first-order logic over finite domains is PSPACE-complete [3]. 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) [5].

CSP provides a general framework in which a wide variety of combinatorial search problems can be expressed in a natural way [6,7]. An instance of 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. Therefore, from a logic approach, CSP is viewed as the model checking problem for $\mathcal{FO}(\exists, \wedge)$. This approach has proven to be very successful, by [8], due to the connection between the logic notion of definability and the complexity of CSP. CSP is NP-hard (actually, it is NP-complete). Indeed, [8] shows

✉ Paqui Lucio
paqui.lucio@ehu.eus

Alex Abuin
aabuin@ikerlan.es

Unai Diaz de Cerio
UDiazCerio@ikerlan.es

Montserrat Hermo
montserrat.hermo@ehu.eus

[1] Computer Languages and Systems, University of the Basque Country, San Sebastián, Spain

[2] Dependable Embedded Systems, Ikerlan Research Center, Mondragón, Spain

that 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 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 special cases of the general problem that can be solved in polynomial time. 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 [9] and its many generalizations. The second type, proposed by [10,11], is to restrict the structure induced by the predicates on the variables.

QCSP is a natural generalization of 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 [12], whereas a good, and quite recent, survey on QCSP and closely related problems is [13]. QCSP is actively studied in artificial intelligence, where it is used to model problems, for example, in non-monotonic reasoning [14] and in planning [15]. Several works such as [16–19] have proposed (superpolynomial or incomplete) algorithms for QCSP over the boolean domain. Quite recent research by [19–22] have started investigations on solving non-boolean QCSP problems. Since QBF can be expressed as a QCSP instance, as showed by [8], QCSP is PSPACE-complete in general. Like in the CSP case, a lot of research is being done nowadays trying to find families of instances that can be decided in polynomial time. It is in this context where a proof system for QCSP, called PS, was introduced by [23]. PS is a slight variant—more efficient in terms of the number of rules—of the proof system previously defined by [24]. 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. As stated in [24], a good understanding of how PS-style proofs are generated provides clues on the very nature of PSPACE-complete problems.

So far, we have discussed the main motivation to formalize PS. Besides, [24] shows that Q-resolution can be simulated in the restriction of PS to the boolean domain. Many of the QBF solvers are based on Q-resolution. The Q-resolution method was introduced by [16]. Since then many different extensions and variants have been proposed, such as long-distance resolution [25,26], QU-resolution [27], and LQU-resolution [28] that 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.

Computer-assisted reasoning has turned out to be a useful tool in a wide range of areas from pure mathematics to smart contracts. Mechanized 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. Therefore, there is genuine value in having mechanized (machine-checked) proofs.

LCF-style proof assistants, such as Agda [29], Coq [30], and Isabelle/HOL [31], are mechanized reasoners based on a small inference kernel of proof rules. Consequently, their implementation relies on a small trusted code base. The basic mechanism for developing machine-checked proofs in the LCF-style (introduced by [32]) consists in invoking a proof rule to be applied to a goal. The tool checks if the rule is applicable and, if so, it automatically generates the subgoals to be proved by the user. In the last years, LCF-style tools have integrated automatic proof search. The user can invoke fully automatic provers (e.g. E [33], SPASS [34], Vampire [35], Z3 [36]) for proving a goal. For utmost reliability, proofs found automatically are translated back into the formalism of the proof assistant and then certified by its trusted code base. LCF-style proof assistants have been successfully used for this task for many years, producing an extensive collection of system formalizations and mechanized proofs. In [37, Sect. 5], the author gives a fairly comprehensive report of the most valuable work done, since the 1980s, in this area. Ringer et al. [38] is an extensive review on developments based on proof assistants for different kinds of software systems. There are many quite recent machine-checked formalizations of checkers or solvers, e.g. the works by [39,40], and many others—such as [41–45]—that also benefit from code generation tools and techniques to obtain executable code from formalization.

Automatic program verifiers—such as ACL2 [46], VCC [47], F* [48], VeriFast [49], Why [50], and Dafny [51]—are dedicated reasoners to verify behavioral properties of programs written in some specific programming language that also work in an interactive way. The basic unit for developing a proof is the assertion of a formula. In order to verify a program or to prove a lemma, the user writes a collection of assertions. The verifier converts assertions into proof obligations that are passed to fully automated provers. Whenever all proof obligations are automatically proved, the verifier (trusts engines and) reports a successful verification. Assertion violations are also reported along with feedback for the user. It has been quite recently shown by [52] 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 have extended their specification language with, among others, 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. Some examples using fixpoint formalizations in Why3 are given by [53]. The first formalization of a rule-based system, using mostly-automatic fixpoint reasoning, was introduced by [52]. Rustan and Leino [54] introduces fixpoint reasoning for Dafny, 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 PS, the machine-checked proofs of its soundness and completeness, and the model checker obtained by automatic code generation. This work has been developed by people with different levels of expertise in formal methods and in industrial software development. One of our aims is to spread formal techniques and related tools in the industrial software development area. As proposed by [41,43], our fully verified checker can serve as a trusted application for testing results of other more efficient, but untrusted, solvers (e.g. QBF-solvers). 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.[1] 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 "A proof system for QCSP" we introduce the proof system PS and its least fixpoint operator. In "Dafny: Language, verifier and IDE" we provide basic notions of the Dafny language and verifier. In "Formalization of the proof system PS in Dafny" we describe the formalization of the proof system PS as an inductive predicate with all the technical details. In "Dafny proofs of soundness and completeness", we explain the main ideas behind the mechanized proofs of soundness and completeness. In "Modular structure" we explain the structure of modules and its dependencies of our formalization, whereas in "Implementation" and "Experience" we respectively give implementation and experience details.

A Proof System for QCSP

In this section we introduce the proof system PS and the least fixpoint of the PS derivability relation. For that, we present definitions for all the necessary basic notions on QCSP (taken from [23,24]). Their Dafny encoding is presented in "Formalization of the Proof System PS in Dafny" where we specify, for each Dafny snippet, the concept that is being encoded.

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 $\mathsf{ar}(R)$ that is an element of $\mathbb{N}$. An atom is an application of a predicate $R(x_1 \ldots x_{\mathsf{ar}(R)})$, where $x_1 \ldots x_{\mathsf{ar}(R)}$ 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.

Definition 1 A *structure* $\mathbf{B}$ on signature σ consists of a finite non-empty *domain* B and an *interpretation* that associates with each symbol $R \in \sigma$ a relation $R^{\mathbf{B}} \subseteq B^{\mathsf{ar}(R)}$.

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 $\mathbf{B}$.

[1] Microsoft Visual Studio project.

Definition 2 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 $\mathbf{B} \models \phi$. In the following example, we illustrate how that 3-QBF problem—i.e. the QBF problem where every clause has exactly three literals—can be expressed as a QCSP.

Example 1 Let us consider the following 3-QBF instance (namely ψ):

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

Let σ be the signature $\{R_{0,3}, R_{1,3}, R_{2,3}, R_{3,3}\}$ and $\mathbf{B}$ the structure on σ with domain $=\{0, 1\}$ such that

$$R^{\mathbf{B}}_{0,3} = \{0, 1\}^3 \backslash \{(0, 0, 0)\}$$
$$R^{\mathbf{B}}_{1,3} = \{0, 1\}^3 \backslash \{(1, 0, 0)\}$$
$$R^{\mathbf{B}}_{2,3} = \{0, 1\}^3 \backslash \{(1, 1, 0)\}$$
$$R^{\mathbf{B}}_{3,3} = \{0, 1\}^3 \backslash \{(1, 1, 1)\}$$

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

$$R^{\mathbf{B}}_{0,3}(x, y, z) = (x \vee y \vee z)$$
$$R^{\mathbf{B}}_{1,3}(x, y, z) = (\neg x \vee y \vee z)$$
$$R^{\mathbf{B}}_{2,3}(x, y, z) = (\neg x \vee \neg y \vee z)$$
$$R^{\mathbf{B}}_{3,3}(x, y, z) = (\neg x \vee \neg y \vee \neg z)$$

where each constraint $R_{i,j}(x, y, z)$ is satisfied by an assignment if and only if the equivalent clause is satisfied by the assignment. For example, $R_{1,3}(x, y, z)$ and the clause $(\neg x \vee y \vee z)$ are satisfied by the same set of assignments. Then, the 3-QBF problem ψ is equivalent to the QCSP instance $(\varphi, \mathbf{B})$ where

$$\varphi = \forall s\, \exists t\, \forall u\, \exists v (R_{2,3}(u, t, s) \wedge R_{1,3}(s, t, v) \wedge R_{1,3}(v, s, t) \wedge R_{0,3}(v, u, s)).$$

In this way, every instance of the 3-QBF problem can be readily translated into an instance of QCSP having the same satisfying assignments. ▲

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 to 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 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*.

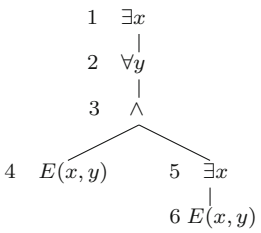

Fig. 1 Formula discussed in Example 2 (from [24])

Example 2 Consider the sentence $\phi = \exists x \forall y (E(x, y) \wedge (\exists x E(x, y)))$ (see Fig. 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 $\{1, \ldots, 6\}$. 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 indices are 4 and 5, and index 3 has one parent whose index is 2.

Definition 3 (*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 $F = \emptyset$.

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. 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 f is a mapping and $v \in B$, we use $f[x \mapsto v]$ to denote the extension or update of f that maps x to v. This notation is also used for multiple updating as $f[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, and also $f[X \mapsto V]$

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

$$\text{(join)}\ \frac{(j,U_j,F_j) \quad (k,U_k,F_k)}{(i,U_j \cup U_k, F_j \bowtie F_k)}\ \text{where } \phi(i) = \phi(j) \wedge \phi(k)$$

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

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

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

Fig. 2 The proof system PS

where X, V respectively represents the tuples $(x_1 \ldots, x_n)$ and $(v_1 \ldots, v_n)$. When f is a mapping from V to B and U is a subset of V, we use $f \restriction U$ to denote the restriction of f to U.

Definition 4 (*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

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

Let (V, F) be a constraint and $U \subseteq V$ with $\{w_1, w_2, \ldots, w_r\} = V \backslash U$. 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 $(\phi, \mathbf{B})$, meaning that $\mathbf{B} \not\models \phi$. The next definition introduces the inference rules of the proof system PS.

Definition 5 (*PS proof system*) A *judgement proof* on a QCSP instance $(\phi, \mathbf{B})$ on signature σ is a finite sequence of judgements that are obtained by application of the inference rules in Fig. 2.
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 worth noting that Definition 3 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. Note also that if $(i, \{\}, \{\})$ can be derived for some index i then $(r, \{\}, \{\})$ can be derived for the root index r using (upward flow).

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 minimal. Obviously, the correctness of the upward flow rule relies on 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:

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 (∀-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 judgements on a QCSP instance $(\phi, \mathbf{B})$. The set $\mathcal{P}(\mathcal{J})$ (all subsets over $\mathcal{J}$) is partially ordered by the $\subseteq$-relation. We define the map $D_{(\phi,\mathbf{B})}$ from $\mathcal{P}(\mathcal{J})$ to $\mathcal{P}(\mathcal{J})$ such that for any $S \in \mathcal{P}(\mathcal{J})$:

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

Given any QCSP instance $(\phi, \mathbf{B})$, the least fixpoint of $D_{(\phi,\mathbf{B})}$ is the set of all derivable judgements according to the fixpoint semantics. The following two examples show how least fixpoint of $D_{(\phi,\mathbf{B})}$ are calculated.

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 $D_{(\phi,\mathbf{B})}$.

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

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

Example 5 Let ϕ be the sentence from Example 2 over signature $\sigma = \{E\}$ with $ar(E) = 2$. Consider ϕ to be indexed as shown in Fig. 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 $\{\{x \mapsto a\}, \{x \mapsto b\}\}$, G is the set of mappings $\{\{x \mapsto a\}\}$, and H is the set of mappings $F_E \restriction \{y\} = \{\{y \mapsto a\}, \{y \mapsto b\}, \{y \mapsto c\}\}$.

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

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 (see [55]), 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 contributions 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 $\mathbf{B} \models \phi$.

Dafny: Language, Verifier and IDE

Dafny [51] is a program verifier that includes a programming language and specification constructs. The Dafny user creates and verifies both specifications and implementations. The Dafny specification language extends first-order logic with algebraic data types, inductive predicates, 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 facilitate the understanding of the rest of the paper.

The basic unit of a Dafny program is the **method**[2]. 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 a ghost method. The body of a lemma is its proof. For lemma proofs, Dafny provides a special notation that is easy to read and understand: *calculations* that were presented by [56]. A calculation in Dafny is a statement that proves a property. This notation was extracted from the *calculational method* introduced by [57], 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 indicated, 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 indicated (in curly brackets after the relationship). Calculations are written inside the environment **calc** .

[2] From now on, we colorize Dafny keywords with different colors similar to Visual Studio code editor.

Dafny also provides built-in immutable types, such as **set**, **multiset**, **map**, and **seq**—which respectively denote the finite collections types of sets, multisets, maps, 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, ... xi: Ti | P(x1,x2,...,xi) • E(x1,x2,...,xi)
```

for defining the set of all values given by the expression `E(x1,x2,...,xi)` for all tuples `(x1,x2,...,xi)` such that `P(x1,x2,...,xi)`.[3]

Dafny also offers user-defined specification constructs (which are ghost), such as **function** and **predicate** that can be defined using well-founded inductive definitions, built-in immutable types, polymorphic algebraic **datatype**s, inductive **predicate**s, etc.

The Dafny specification constructor **inductive predicate** (also called *extreme predicates*) was introduced by [54] and allows for the definition of a predicate as an extreme solution: a least fixpoint of a set of recursive rules. Inductive predicates are essential to formally define the set of judgements 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 this 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, Dafny verifies the monotonicity of `P`, by checking 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 are given by [54]. In "Formalization of the Proof System PS in Dafny" we introduce an inductive predicate (`is_derivable`) and prove an inductive lemma (`models_Lemma`).

For defining variables in methods, functions and proofs, Dafny includes a *let-such-that statement*: **var** `x :| P` that looks like a variable declaration but it includes a boolean expression `P` after the so-called *Hilbert epsilon operator* or *choose operator* `:|`. A statement **var** `x :| P` can be read as assign to `x` any value such that it satisfies `P`. The verifier must be able to prove that there exists a value which

[3] For easy reading, in the Dafny code snippets, we show the usual mathematical symbols, instead of real Dafny notation. For example, we show • for :: (such that), ∪ for union instead of +, ⊆ for set inclusion instead of <=, also for the logical symbols and quantifiers, for example `&&` is shown as ∧ and `forall` as ∀, etc.

meets the condition P, but not to construct it. Then, for verification purposes, the variable x will stand for any value that fulfils P. Consequently, not every let-such-that declarations is directly compilable into executable code. [58] discuss implementation issues of this operator in the language Dafny. [59] provides an example of non-compilable function for which a compilable version is constructed. In "Implementation" and "Experience" we discuss two different problems related with a let-such-that declaration in our formalization and their solution. Our first problem is related to the generation of executable code, and the second one is caused by non-determinism in the proof of a lemma. Our formalization contains several occurrences of let-such-that declarations but all, except the two problematic ones, are in lemma proofs[4] where this non-constructive assignment is natural and useful.

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 information such as the locations (of the properties) related to the error. The interested reader is referred to [60] for further information on the several ways that the 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.

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.

[4] Executable code is not generated for proofs.

A structure (see Definition 1 and Fig. 3) is given by a triple formed by a signature, a domain (i.e. a non-empty finite set), and an interpretation that is a map from the names in the signature to relations on the domain of the arity determined in the signature.

Note that we name by Structure both the datatype and its unique constructor. The type variable 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, wfStructure(B) denotes the non-emptiness of the domain of B and also that every relation symbol r is interpreted in B by sequences whose length is exactly the arity of r.

In Fig. 4, 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.

Note that wfFormula(S,phi) denotes that the formula phi is well-formed with respect to the signature S, that is if the number of parameters of all its atoms coincides with its arity.

A well-formed QCSP instance (see Definition 2) consists of a well-formed structure, a well-formed formula with symbols in the signature of the structure that must be a sentence.

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

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{$[a, a]$, $[a, b]$, $[a, c]$, $[b, a]$}]) that represents the QCSP-instance of Example 3, then wfQCSP_Instance(phi, B) is True.

We also declare judgements and the predicate for checking their well-formedness (see Fig. 5)

A (well-formed) judgement on a (well-formed) QCSP instance (ϕ, **B**), according to Definition 3, is a triple formed by an index i in the set of indexes 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 indices in the nodes of a given formula (seen as a tree, see Fig. 1). In our formalization, for easy access to formula nodes, indices 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

```
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.Keys ∧
                          ∀ t • t in B.I[r] ⟹ |t| = B.Sig[r])
}
```

Fig. 3 Structures in Dafny

```
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) = {} }
```

Fig. 4 CPL Formulas in Dafny

```
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>)
{
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)
}
```

Fig. 5 Judgements in Dafny

that the function `FoI` preserves the well-formedness property with respect to the signature of `phi`.

The inference rules in PS rely upon applying the operations join, projection, and dual-projection on the sets of valuations included in the derived judgements (the component `F` in the datatype). In the course of formalizing PS, we define[5] the three predicates on judgements(see Fig. 6) that correspond with the three operations on mappings (i.p. valuations) in Definition 4.

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

```
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)
}

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])
)
}
```

Fig. 6 Predicates on judgements

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. right-hand) child. If it has only one child, it is `j.i++[0]`. The expression `projectVal(f,U)` represents the projection of `f` on `U`. 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 predicates `is_join`, `is_projection`, and `is_dualProjection`, and `is_upwardFlow` respectively enable the encoding of the inference rule (join), (projection), (∀-elimination), and (upward flow), which are given in Definition 5.

In Fig. 7 inductive predicate `is_derivable` defines, in a natural way, the least fixpoint of the derivability relation induced by the five rules in Definition 5.

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 superscript 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 argu-

```
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)
)
}
```

Fig. 7 The inductive predicate `is_derivable`

ments `phii.par`, this gives a tuple that is checked to belong to the interpretation of relation `phii.rel` in the structure `B`.

these meta-logical results we use the predicate (see Fig. 8) states whether a QCSP instance `(B,f)` is a model of a formula `phi`.

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)))
}
```

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

Given a valuation `h` and a judgement `j` on a QCSP instance `(phi,B)`, predicate `valuationModel` denotes 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 `freeVar(FoI(j.i,phi,B.Sig))\j.V` represents $\mathsf{freeVar}(\phi(i))\backslash$`j.V` and the function `existSq` enables the

```
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)
}
```

Fig. 8 The predicate `models`

```
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)
}
}
```

Fig. 9 The inductive lemma `models_Lemma`

existential closure (for its definition see Fig. 18). 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 $\mathsf{freeVar}(\phi(i))\backslash \mathtt{j.V} = \{x_1, \ldots, x_n\}$.

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

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* $\mathsf{freeVar}(\phi(i))\backslash V$. *For all* $h : V \to B$ *it holds that* $\mathbf{B}, h \models \exists v_1 \ldots \exists v_n \phi(i)$ *implies* $h \in F$.

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

Since `models_Lemma` (see Fig. 9) is an inductive lemma, its proof performs induction on the construction of the least fixpoint generated by the inductive (extreme) predicate `is_derivable`. Dafny automatically enables this kind of induction for inductive lemmas, and automatically constructs the induction hypothesis from the recursive calls to `models_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

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;
        }
```

Fig. 10 The auxiliary lemma `projection_Lemma`

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 any 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 auxiliary lemma in Fig. 10.

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 postcondition 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[6]

We define the variable `phii` (i.e. $\phi(i)$) and the three sets of variables `W`, `Y` and `X` occurring in $\phi(i)$ and in the judgements `j` and `j'`. Next, we prove that any valuation h such that $B, h \models \exists X(\phi(i))$ belongs to `j.F`. This is the meaning of the ∀-`ensures` in the code, whose proof is inside the curly brackets. From the hypothesis, since `X` = `Y` ∪ `W`, we prove 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 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 $h[Y := Z]$. Then, by hypothesis `(H2)`, we prove that $h[Y \mapsto Z] \in j'.F$. Therefore, its projection $h[Y \mapsto Z] \upharpoonright j.V$ (which is proved to coincide with the map-

[6] In snippets, commented assertions are (true) assertions that Dafny automatically infers, hence Dafny does not need them as hints. They are included in proofs mainly for human-readability. The user could check the validity of each assert by uncommenting it. They are also very useful for re-using or refactoring proofs.

```
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);
}
```

Fig. 11 The soundness theorem

```
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)
}
```

Fig. 12 The function `canonical_judgement`

ping h) should belong to $j.V$. Since (by hypothesis `H1`) $j.F$ is the set of all projection on $j.V$ of all valuations in $j'.F$, then $h \in j.F$ is proved.

The other three inductive cases in the proof of the `models_Lemma`—for derivability using (join), (∀-elimination), and (upward flow)—follows similar lines as the case for (projection) and their code is omitted.

In Fig. 11, the `soundness_Theorem` of the proof system PS can be easily proved by calling the previous `models_Lemma`.

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 model `phi`.

Completeness, i.e. the backward direction of Theorem 1, is encoded in the following `completeness_Theorem` that 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 with each index `i` of the formula `phi`. We call it the *canonical judgement*. It is recursively defined by the function in Fig. 12.

Note that `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 indices of the subformulas of `phi,phi1,phik`, in the succeeding recursive

```
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)
⊞{...}

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))
}
```

Fig. 13 The lemma `canonical_judgement_Lemma`

```
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 := 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)) = {};
}
```

Fig. 14 The completeness theorem

definition given by a **match** statement, are correct. Consequently, Lemma 2 is encoded in the Dafny lemma (Fig. 13) that uses the function `setOfValModels` 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`. The proof consist of 50 lines of code (that we do not show here), and produces not only recursive calls (for the induction hypothesis), but also calls to three auxiliary lemmas that formalize interesting 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). The proofs of these three lemmas are also non-trivial, each has approximately 30 lines of code. They also call simpler lemmas that prove some elementary properties of the projection or extension of a valuation. Most of the effort was not made in the proof of `canonical_judgement_Lemma` (whose inductive schema is large but easy), but in the three auxiliary lemmas that prove equalities between different sets of valuations. However, since valuations, sets, projections, etc. have a natural and easy formalization in the Dafny language, our formalized proofs essentially follows the guidelines of the pen-and-paper proofs. In "Experience", we provide some figures about `canonical_judgement_Lemma`.

Next, the `completeness_Theorem`, (see Fig. 14) 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 canonical judgement for `([], phi, B,)`.

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

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 crucial for refactoring and reusing a large formalization. Moreover, in our experience, modularity is clearly helpful during the development phase. In this section, we give an idea of the whole encoding by describing how it is structured using modules.

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 also clauses **include** to include one module in another. In the head of modules, Dafny allows clauses **import** and **export** and different qualifiers.

In the case of **import**, the qualifier **opened** allows the names of the imported units to be used (in the importing module) 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 lists. In addition, the qualifiers `provides` and `reveals` allow 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 may not 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 [61].

Our formalization is structured in seven modules whose dependencies are described in Fig. 15. 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 "Implementation".

Module `Utils` contains a few auxiliary concepts and properties on sets, sequences and maps that are of general utility. In Fig. 16 we show part of it.

The export list of module `Utils` reveals a function `setOf` for the set of elements of a sequence and a predicate `noDups` for deciding whether a given sequence has duplicates, both have no ensures clause. However, it provides (but does not reveal) the function `allMaps` (see "Formalization of the Proof System PS in Dafny"), hence importer modules know its two ensure clauses, but not its body. The two ensures clauses of the function `allMaps` provides a partial specification of it that is automatically verified (hence, the code

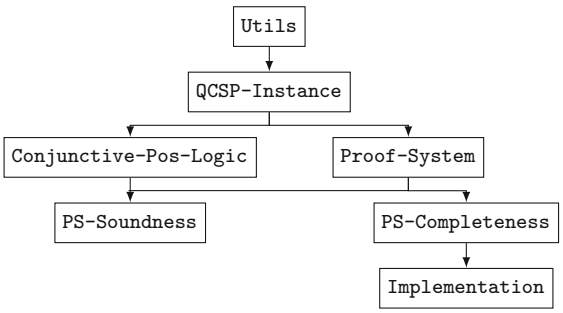

Fig. 15 Module dependencies (w.r.t. **include** clauses). The module at the head of each arrow includes the one at the tail of the arrow

of the function does not need for annotations). Moreover, this partial specification suffices to verify most of methods involving a call to `allMaps`. Module `Utils` provides a lemma `allMaps_Correct_Lemma`, which complements the first ensures clause of `allMaps` with the backward implication. This lemma has a non-trivial proof, hence we decided to prove it separately from the code of the the function. We call this lemma only when the complete specification of `allMaps` is required to verify some property. Module `Utils` also provides two lemmas on operations over sets of maps.

In "Formalization of the Proof System PS in Dafny" and "Dafny Proofs of Soundness and Completeness" we explained the most relevant components 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. In Fig. 17 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`.

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 module `Conjunctive-Pos-Logic` is to provide the properties of Conjunctive Positive Logic that we need to prove soundness. Indeed, all are properties about the models of formulas of the form $\exists x_1 \ldots \exists x_n \phi$, which is written in Dafny as `existSq(W,phi)` where `W` is the sequence $[x_1 \ldots x_n]$. The function `existSq` is defined and exported by module `Conjunctive-Pos-Logic`, see Fig. 18.

As an example of the kind of properties (about models of `existSq`-formulas) that module `Conjunctive-Pos-Logic` provides to module `PS-Soundness`, we show (in Fig. 18)

```
module Utils {
export reveals setOf, noDups
       provides allMaps,allMaps_Correct_Lemma,ExtMap_Lemma,
                ProjectMap_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[]}
⊞{...}
...
}
```

Fig. 16 Partial view of Module `Utils`

```
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..])
⊞{...}

... // 15 more 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)))
⊞{...}
```

Fig. 17 Partial view of Module `QCSP_Instance`

```
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
⊞{...}

... // 12 more lemmas
}
```

Fig. 18 One of the lemmas exported from `Conjunctive-Pos-Logic` to prove the soundness of proof system PS

the specification part of `existSq_And_Lemma`. This lemma states 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 Fig. 18) assist in the task of proving the lemma `models_Lemma` that is crucial in the soundness proof, as explained in "Dafny Proofs of Soundness and Completeness". Since `models_Lemma` (or Lemma 1) refers to an existentially closed formula, the metalogical properties in module `Conjunctive-Pos-Logic` are mainly related to existentially closed formulas.

Implementation

On the basis of our formalization of the proof system PS, and employing Dafny's automatic code generation facilities, we have implemented a verified model checker for Conjunctive Positive Logic. Dafny and most of the popular verification tools (Isabelle/HOL, Why3, etc.) are equipped with code generators that are able to translate into executable code a sublanguage of their formal specification language. Some formally verified applications (e.g. [62]) have been developed while keeping in mind not only verification, but also direct code generation (sometimes even code efficiency). As explained in [63], the restrictions of the specification sublanguage often lead to complications in formal definitions and proofs. We have taken the different approach of developing the (above explained) formalization while having in mind only verification purposes. Once the verification phase was successfully completed, we have refined the verified formalization to exploit the Dafny automatic extraction of executable code.

Our refinement basically consists in two task: (1) to convert into executable some non-executable specification units and (2) to make usable the C# classes that Dafny generates for datatypes. A similar work for a larger formalization in Isabelle/HOL is explained in [63]. 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 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` must also 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 an expression of the form `∀ i: nat • P(i)` that appears in the 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.

```
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)
```

The interface of our web application asks the user to successively provide the different components of a QCSP instance, then it checks the well-formedness of the given QCSP instance.When a well-formed QCSP instance $(\mathbf{B}, \phi)$ is given, the application returns the result of whether $\mathbf{B} \models \phi$. For that, it computes the canonical judgement for $\mathbf{B} \models \phi$. If it is empty, it answers "no", otherwise the answer is "yes". Therefore, 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 those just transformed into non-ghost. We made this until Dafny does not anymore complain that"function calls are allowed only in specification context (consider declaring the function as function method)". Dafny marks the affected calls and shows those 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 two more predicate and function 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, namely `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` includes a non-compilable let-such-that statement **var** `a :| a` **in** `s`, where `s` is a set. It 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`. We verify the equivalence of each method with the original function. Actually the contract of each method `compute_f` specifies that it computes the function `f`. For example, the contracts of `compute_canonical_judgement` and `compute_allMaps` are:

After all these changes, all proofs are still successfully checked by Dafny. Then, 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`. For exam-

ple, the `datatype Formula` has four constructors in the 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 use 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 is 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`preconditions are:

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

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

The web application is a graphical input interface to make use of the .NET library that is generated by the Dafny compiler. It firstly check that the `QCSP_Instance` typed by the user is well formed by calling the corresponding library method. Then, it calls the library method that checks its satisfiability, and reports the result to the user.

The web application code consists of 5 pages: index, signature, domain, interpretation and formula. This code has a total of 660 lines. 48% of the code defines the Graphic User Interface and its functionalities. 33% of the code defines the business logic. Finally, only 19% of the code makes calls to the .NET library.

Experience

Our formalization and implementation has been developed in the Dafny IDE [60] that lends itself to increasing user productivity. The Dafny IDE is an extension of Microsoft Visual Studio (VS) that runs the program verifier in the background and provides design-time feedback.

When an attempted verification fails—e.g an assertion violation occurs, a loop or recursion could not terminate, an invariant fails to be ensured on entry or to be maintained by the loop, etc.—a red dot (and a red squiggly line) indicate the return path along which the error is reported. The error panel at the bottom of the screen shows the error message, which also appears as hover text for the red squiggly line. The error panel also lists the source locations related to the error. These source locations are also marked by a violet squiggly line with associated hover text.

Examples of error source locations are: a specific postcondition (that cannot be verified), a user-defined predicate that is called in an (violated) assertion, the precondition of a method that is not verified by a method call, the termination measure of a loop or a recursive definition, etc.

In addition, Dafny could provide more information about the failing situation: the user can explore the (potential) counterexample produced by the solver. By clicking on a red dot, the Dafny IDE displays in a screen a table of states that describe the counter example and, simultaneously, a collection of blue dots appear in the program text. There is state information associated with each blue dot. The user can click on a blue dot to select a particular state and trace the control path leading to the error.

The main features of the Dafny IDE are well described in [60].

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.

Dafny supports a number of proof features traditionally only found in LCF-style 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 less precise form. Mechanized 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` that mainly contain logical equivalences that are usually assumed. Moreover, it may happen that the process of proving some of these properties raises some issue improperly 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. Particularly noteworthy are preconditions that the programmer assumes, but she (or he) does not make explicit in the specification. Along the development of our proof many subtle details have been fixed. For example, we forgot to specify, as part of the predicate `wfStructure`, that the domain of the given structure must be non-empty. We realized that when we were not able to proof lemma `existSqSem_Lemma` in module `Conjunctive-Pos-Logic`.

The lemma `existSqSem_Lemma` asserts that if $\mathbf{B}, f \models \exists x_1 \ldots \exists x_n \alpha$ (where $x_i \notin f.Keys$ for all $1 \leq i \leq n$), then there exists $v_1, \ldots, v_n \in B$ such that

$$\mathbf{B}, f[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \models \alpha.$$

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

Module name	Lines	Datatypes	Functions	Lemmas	Methods	PO	Secs
`Utils`	65	0	4	3	0	39	3
`QCSP-Instance`	358	2	8	19	8	697	17
`Proof-System`	218	1	7	4	1	469	14
`Conj-Pos-Logic`	363	0	2	13	0	1042	46
`PS-Soundness`	288	0	1	8	0	1004	32
`PS-Completeness`	231	0	4	5	1	698	17
`Implementation`	204	0	0	4	7	256	10
TOTAL	1727	3	26	56	17	4205	139

We prove it by induction on the sequence of variables $x_1, \ldots, x_n$. For that, one could choose any variable x_i. For avoiding unnecessary details here, we suppose that we choose x_1. Then, from $\mathbf{B}, f \models \exists x_1(\exists x_2 \ldots \exists x_n \alpha)$ and the semantics of $\exists$, we get that there exists $v_1 \in B$ such that $\mathbf{B}, f[x_1 \mapsto v_1] \models \exists x_2 \ldots \exists x_n \alpha$. From the latter, by induction hypothesis, there exists $v_2, \ldots, v_n \in B$ such that

$$\mathbf{B}, f[x_1 \mapsto v_1][x_2 \mapsto v_2, \ldots, x_n \mapsto v_n] \models \alpha.$$

Hence, by composing the mapping updates the proof is done. The problem appeared in obtaining v_1, since we defined it in Dafny as any value v *such that* $v \in B$ and $\mathbf{B}, f[x_1 \mapsto v] \models \exists x_2 \ldots \exists x_n \alpha$. Then, Dafny complains: "*Error: cannot establish the existence of LHS values that satisfies the such-that predicate.*" When we wrote, in the line above the such-that clause, the assumed assertion[7] `assume B ≠`, the Dafny error disappeared revealing our error: a well-formed structure should not have an empty domain.

Another noteworthy 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 that the quantified variable (in the formula) does not always belong to the set of variables of the judgement. The way we interacted with Dafny in this detection is very similar to the above explained error detection, though technicalities of the problem are more complex.

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 Hilbert's 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.

Our formalization is structured in seven modules. 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 (PO i.e. queries discharged to Z3) and the seconds required to verify each module by Dafny 2.3.0 for Windows (x64) running on a processor i5-7500 CPU at 2.60GHz with 16 GB of RAM.

The proof of lemma `existSq_Distr_And_Lemma` takes about 17 seconds, which is the most costly proof, for solving 152 PO. However, the largest set of PO, which consists of 423, is generated by `models_Lemma_h` and it is proved in 0.42 seconds.

To give some global figures about the 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, 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 closest to 250, exactly 218 PO, and it is verified in 9 s.

The 'proof obligations to lines' ratio is (2.4:1), i.e. Dafny generates 2.4 PO for every 1 line of code. A large share of the proof obligations are automatically generated by Dafny

[7] In order to help the user in the process of constructing proofs, Dafny allows to introduce assumed assertions P. This allows the user to check whether P is the condition that Dafny needs to complete the proof. Dafny considers non-verified any proof with an assume clause.

to check, for example, that (partial) expressions are defined or recursive definitions and loops do terminate. On one hand, expressions in Dafny are partial i.e. they are undefined for some values of their variables. For example, given a sequence `s`, the expression `s[i]` is defined only if `i` is an index into the sequence `s` range. Similarly, for a user-defined function `F` with formal parameter `x`, a function call `F(a)` is defined only if `a` satisfies the user-defined precondition (of function `F`) where `x` is substituted by `a`. On the other hand, Dafny automatically generates POs to check that a termination metric is bounded by 0 and it decreases at each step of the loop or at each recursive call. The termination metric can have been inferred by Dafny (no lines in code) or specified by the user (usually, 1 line in code).

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

Acknowledgements We are greatly indebted to Rustan Leino for the time and effort he has kindly devoted to us. This work has been carried out thanks to his thoughtful and stimulating suggestions and his considerable research assistance. Our work did specially benefit from his in-person tutorial of the inductive-predicate features and his advice for how to formulate the definitions in Dafny. We would also like to thank Hubie Chen for his support during the early developments of this work. We are also very grateful to the anonymous reviewers for their many helpful comments and suggestions that led to substantial improvements in the presentation of the paper.

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.

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen D, editor. Logics of programs. Berlin: Springer; 1982. p. 52–71. https://doi.org/10.1007/BFb0025774.
2. Queille JP, Sifakis J. Specification and verification of concurrent systems in cesar. In: Dezani-Ciancaglini M, Montanari U, editors, International Symposium on Programming; 1982. Berlin: Springer, p. 337–351. https://doi.org/10.1007/3-540-11494-7_22.
3. Stockmeyer LJ. The Complexity of Decision Problems in Automata Theory and Logic. Project MAC: MAC TR. Massachusetts Institute of Technology; 1974.
4. Vardi MY. The complexity of relational query languages. STOC '82. New York: Association for Computing Machinery; 1982. p. 137–46. https://doi.org/10.1145/800070.802186.
5. Martin B. Dichotomies and duality in first-order model checking problems. CoRR. 2006. http://arxiv.org/abs/cs/0609022.
6. Creignou N, Khanna S, Sudan M. Complexity classifications of Boolean constraint satisfaction problems, volume 7 of SIAM monographs on discrete mathematics and applications. Philadelphia: Society for Industrial and Applied Mathematics; 2001. https://doi.org/10.1137/1.9780898718546.
7. Dechter R. Constraint processing. Burlington: Morgan Kaufmann Publishers Inc.; 2003. https://doi.org/10.1016/B978-1-55860-890-0.X5000-2.
8. Chen H. A rendezvous of logic, complexity, and algebra. ACM Comput Surv. 2009;42(1):21–232. https://doi.org/10.1145/1592451.1592453.
9. Schaefer TJ. The complexity of satisfiability problems. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78; 1978. New York: ACM, p. 216–226. https://doi.org/10.1145/800133.804350.
10. Grohe M. The complexity of homomorphism and constraint satisfaction problems seen from the other side. J ACM. 2007;54:1–24.
11. Marx D. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. J ACM. 2013;. https://doi.org/10.1145/2535926.
12. Martin B. First-order model checking problems parameterized by the model. In: Beckmann A, Dimitracopoulos C, Löwe B, editors. Logic and theory of algorithms. Berlin: Springer; 2008. p. 417–27. https://doi.org/10.1007/978-3-540-69407-6_45.
13. Martin B. Quantified constraints in twenty seventeen. In: Krokhin A, Zivny S, editors. The constraint satisfaction problem: complexity and approximability, volume 7 of Dagstuhl follow-ups; 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, p. 327–346. https://doi.org/10.4230/DFU.Vol7.15301.327.
14. Egly U, Eiter T, Tompits H, Woltran S. Solving advanced reasoning tasks using quantified boolean formulas. In: Kautz HA, Porter BW, editors. In: Proceedings of the 17th national conference on artificial intelligence and 12th conf. on innovative applications of artificial intelligence; 2000. AAAI Press/The MIT Press, p. 417–422. http://www.aaai.org/Library/AAAI/2000/aaai00-064.php.
15. Rintanen J. Constructing conditional plans by a theorem-prover. J Artif Intell Res. 1999;10(1):323–52 http://dl.acm.org/citation.cfm?id=1622859.1622870.
16. Buning HK, Karpiński M, Flogel A. Resolution for quantified Boolean formulas. Inf Comput. 1995;117(1):12–8. https://doi.org/10.1006/inco.1995.1025.
17. Cadoli M, Schaerf M, Giovanardi A, Giovanardi M. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. J Autom Reason. 2002;28(2):101–42. https://doi.org/10.1023/A:1015019416843.
18. Giunchiglia E, Narizzano M, Tacchella A. Backjumping for quantified Boolean logic satisfiability. Artif Intell. 2003;145(1–2):99–120. https://doi.org/10.1016/S0004-3702(02)00373-9.
19. Williams R. Algorithms for quantified boolean formulas. In: Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms, SODA '02; 2002. Society for Industrial and Applied Mathematics, p. 299–307. http://dl.acm.org/citation.cfm?id=545381.545421.
20. Bordeaux L, Monfroy E. Beyond NP: arc-consistency for quantified constraints. In: Van Hentenryck P, editor. Principles and practice of constraint programming-CP. Berlin: Springer; 2002. p. 371–86. https://doi.org/10.1007/3-540-46135-3_25.
21. Gent IP, Nightingale P, Rowley A, Stergiou K. Solving quantified constraint satisfaction problems. Artif Intell. 2008;172(6):738–71. https://doi.org/10.1016/j.artint.2007.11.003.
22. Mamoulis N, Stergiou K. Algorithms for quantified constraint satisfaction problems. In: Proceedings of CP'04, volume 3258 of LNCS, 2004. Springer, p. 752–756.
23. 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. http://hdl.handle.net/11705/PROLE/2017/017.

24. Chen H. Beyond Q-resolution and prenex form: a proof system for quantified constraint satisfaction. Logic Methods Comput Sci. 2014;. https://doi.org/10.2168/LMCS-10(4:14)2014.
25. Balabanov V, Jiang J-HR. Unified QBF certification and its applications. Formal Methods Syst Des. 2012;41(1):45–65. https://doi.org/10.1007/s10703-012-0152-6.
26. 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); 2002. p. 442–449. https://doi.org/10.1145/774572.774637.
27. Van Gelder A. Contributions to the theory of practical quantified boolean formula solving. In: Milano M, editor. Principles and Practice of Constraint Programming-18th International Conference, CP 2012, Proceedings, volume 7514 of lecture notes in computer science, 2012. Springer, p. 647–663. https://doi.org/10.1007/978-3-642-33558-7_47.
28. Balabanov V, Widl M, Jiang J-HR. QBF resolution systems and their proof complexities. In: Sinz C, Egly U, editors. Theory and applications of satisfiability testing-SAT 2014. Cham: Springer International Publishing; 2014. p. 154–69.
29. 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; 2009. Springer, p. 73–78. https://doi.org/10.1007/978-3-642-03359-9_6.
30. The Coq Development Team. The Coq proof assistant. https://coq.inria.fr.
31. Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL—a proof assistant for higher-order logic, volume 2283 of LNCS. Berlin: Springer; 2002.
32. Gordon M, Milner R, Wadsworth CP. Edinburgh LCF: a mechanised logic of computation, volume of 78 lecture notes in computer science. Berlin: Springer; 1979.
33. Schulz S. System description: E 1.8. In: McMillan KL, Middeldorp A, Voronkov A, editors, Logic for programming, artificial intelligence, and reasoning-19th international conference, LPAR-19, Proceedings, volume 8312 of lecture notes in computer science; 2013. Springer, p. 735–743. https://doi.org/10.1007/978-3-642-45221-5_49.
34. Weidenbach C, Dimova D, Fietzke A, Kumar R, Suda M, Wischnewski P. SPASS version 3.5. In: Schmidt RA, editor, Automated Deduction-CADE-22, 22nd International Conference on Automated Deduction, Proceedings, volume 5663 of lecture notes in computer science; 2009. Springer, p. 140–145. https://doi.org/10.1007/978-3-642-02959-2_10.
35. Riazanov A, Voronkov A. The design and implementation of VAMPIRE. AI Commun. 2002;15(2–3):91–110 http://content.iospress.com/articles/ai-communications/aic259.
36. de Moura L, Bjørner N. Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J, editors. Tools and algorithms for the construction and analysis of systems, 14th international conference, TACAS 2008, volume 4963 of lecture notes in computer science; 2008. Springer, p. 337–340.
37. Blanchette JC. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi A, Myreen MO, editors. Proceedings of the 8th ACM SIGPLAN international conference on certified programs and proofs, CPP. ACM; 2019. p. 1–13. https://doi.org/10.1145/3293880.3294087
38. Ringer T, Palmskog K, Sergey I, Gligoric M, Tatlock Z. QED at large: a survey of engineering of formally verified software. Found Trends Program Lang. 2019;5(2–3):102–281. https://doi.org/10.1561/2500000045.
39. Blanchette JC, 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; 2017. p. 4786–4790. https://doi.org/10.24963/ijcai.2017/667.
40. Schlichtkrull A. Formalization of the resolution calculus for first-order logic. J Autom Reason. 2018;61(1–4):455–84. https://doi.org/10.1007/s10817-017-9447-z.
41. Esparza J, Lammich P, Neumann R, Nipkow T, Schimpf A, Smaus J-G. A fully verified executable LTL model checker. In: Sharygina N, Veith H, editors. Computer aided verification. Berlin: Springer; 2013. p. 463–78.
42. Fleury M. Optimizing a verified SAT solver. In: Badger JM, Rozier KY, editors. NASA Formal Methods-11th International Symposium, NFM 2019, Proceedings, volume 11460 of lecture notes in computer science; 2019. Springer, p. 148–165. https://doi.org/10.1007/978-3-030-20652-9_10.
43. Maric F. Formal verification of a modern SAT solver by shallow embedding into Isabelle/Hol. Theor. Comput. Sci. 2010;411(50):4333–56. https://doi.org/10.1016/j.tcs.2010.09.014.
44. Oe D, Stump A, Oliver C, Clancy K. versat: a verified modern SAT solver. In: Kuncak V, Rybalchenko A, editors. Verification, Model Checking, and Abstract Interpretation-13th International Conference, VMCAI 2012, Proceedings, volume 7148 of lecture notes in computer science; 2012. Springer, p. 363–378. https://doi.org/10.1007/978-3-642-27940-9_24.
45. Schlichtkrull A, Blanchette JC, 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); 2019. Association for Computing Machinery, p. 152–165. https://doi.org/10.1145/3293880.3294100.
46. Kaufmann M, Manolios P, Moore JS. Computer-aided reasoning: an approach. Advance formal methods. Dordrecht: Kluwer Academic Publishers; 2000. https://doi.org/10.1007/978-1-4615-4449-4.
47. 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: Berghofer S, Nipkow T, Urban C, Wenzel M, editors. Proceedings of theorem proving in higher order logics: 22nd international conference, TPHOLs, Munich, Germany, August 17-20; 2009. Springer, p. 23–42. https://doi.org/10.1007/978-3-642-03359-9_2.
48. Swamy N, Chen J, Fournet C, Strub P-Y, Bhargavan K, Yang J. Secure distributed programming with value-dependent types. J Funct Programm. 2013;23(4):402–51. https://doi.org/10.1017/S0956796813000142.
49. Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru MG, Havelund K, Holzmann GJ, Joshi R, editors. NASA Formal Methods. Berlin: Springer; 2011. p. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4.
50. Filliâtre J-C, Paskevich A. Why3—where programs meet provers. In: Felleisen M, Gardner P, editors. Programming languages and systems—22nd European Symposium on Programming, ESOP 2013, volume 7792 of lecture notes in computer science; 2013. Springer, p. 125–128. https://doi.org/10.1007/978-3-642-37036-6_8.
51. Rustan K, Leino M. Dafny: an automatic program verifier for functional correctness. In: Clarke EM, Voronkov A, editors. Logic for programming, artificial intelligence, and reasoning, volume 6355 of lecture notes in computer science. Berlin: Springer; 2010. p. 348–70.
52. Clochard M, Filliâtre J-C, Marché J-C, Paskevich A. Formalizing Semantics with an Automatic Program Verifier. Berlin: Springer International Publishing; 2014. p. 37–51. https://doi.org/10.1007/978-3-319-12154-3_3.

53. Bobot F, Filliâtre J-C, Marché C, Paskevich A. Let's verify this with Why3. Softw Tools Technol Transf (STTT). 2015;17(6):709–27. https://doi.org/10.1007/s10009-014-0314-5.
54. Rustan K, Leino M. Well-founded functions and extreme predicates in Dafny: a tutorial. In: Konev B, Schulz S, Simon L, editors. IWIL-2015. 11th international workshop on the implementation of logics, volume 40 of EPiC series in computing; 2016. EasyChair, p. 52–66. https://doi.org/10.29007/v2m3.
55. Tarski A. A lattice-theoretical fixpoint theorem and its applications. Pac J Math. 1955;5(2):285–309 https://projecteuclid.org:443/euclid.pjm/1103044538.
56. Rustan K, Leino M, Polikarpova N. Verified calculations. In: Cohen E, Rybalchenko A, editors. Verified software: theories, tools, experiments—5th international conference, VSTTE 2013, revised selected papers, volume 8164 of lecture notes in computer science; 2014. Springer, p. 170–190. https://doi.org/10.1007/978-3-642-54108-7_9.
57. Backhouse R, editor. The calculational method, volume 53 of information processing letters. New York: Elsevier; 1995. https://doi.org/10.1016/0020-0190(94)00212-H.
58. Rustan K, Leino M. Compiling Hilbert's epsilon operator. In: Fehnker A, McIver A, Sutcliffe G, Voronkov A, editors. LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning-Short Presentations, volume 35 of EPiC Series in Computing; 2015. EasyChair, p. 106–118. https://doi.org/10.29007/rkxm.
59. Rustan K, Leino M. Dafny power user: iterating over a collection. manuscript krml 275; 2020. https://leino.science/papers/krml275.html.
60. Rustan K, Leino M, Wüstholz V. The Dafny integrated development environment. In: Dubois C, Giannakopoulou D, Méry D, editors. Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, volume 149 of electronic proceedings in theoretical computer science; 2014. Open Publishing Association, p. 3–15. https://doi.org/10.4204/eptcs.149.2.
61. Rustan K, Leino 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; 2018. p. 185–202. https://doi.org/10.1007/978-3-319-98047-8_12.
62. Thiemann R, Sternagel C. Certification of termination proofs using CeTA. In: Berghofer S, Nipkow T, Urban C, Wenzel M, editors, Theorem proving in higher order logics, 22nd international conference, TPHOLs 2009, proceedings, volume 5674 of lecture notes in computer science; 2009. Springer, p. 452–468. https://doi.org/10.1007/978-3-642-03359-9_31.
63. Lochbihler A, Bulwahn L. Animating the formalised semantics of a Java-Like language. In Marko C, van Eekelen JD, Geuvers H, Schmaltz J, Wiedijk F, editors. Interactive theorem proving-second international conference, ITP 2011, proceedings, volume 6898 of lecture notes in computer science; 2011. Springer, p. 216–232. https://doi.org/10.1007/978-3-642-22863-6_17.