Towards the Automatic Verification of QCSP Tractability Results

Alex Abuin[1], Hubie Chen[2], Montserrat Hermo[3], and Paqui Lucio[3] ⋆

[1] Dependable Embedded Systems, Ik4-Ikerlan Research Center, Mondragón, Spain
[2] Universidad del País Vasco and IKERBASQUE Basque Foundation for Science, San Sebastián, Spain
[3] Departamento LSI, Universidad del País Vasco, San Sebastián, Spain

Abstract. We deal with the quantified constraint satisfaction problem (QCSP) which consists in deciding, given a structure and a first-order sentence built from atoms, with conjunction and quantification, whether or not the sentence is true on the structure. We study a known proof system which has been used to derive QCSP tractability results. Our contribution is to formalize this proof system into an automatically verified theory, so that it can be used (in a near future) as a basis for automatically verify tractability results.

1 Introduction

Automated Verification: Automatic program verifiers prove properties via the use of automatic theorem provers, such as SMT solvers. They have turned out to be useful tools for formalizing programming languages semantics, compilers, virtual machines, program logics, operating systems and, in general, symbolic computation engines. Such formalizations are a mix of definitions and proved properties (or lemmas), which all together provide a computer-checked proof of the correctness (to some extent) of the engine. Very often, such formalizations are very long and complicated and therefore difficult to check by hand, so there is genuine value in having a computer-checked proof. This will be especially true if it eases the formalization of future more complicated, but very similar, proofs to those already proved. That is main motivation of our work: tractability results derived from the QCSP proof system we present here involve large (often inductive) proofs with thorough details. Moreover, as extensions of the system are in prospect, we plan to use the formalization as a certified basis for the next extensions, that will be obtained as re-factorizations of the initial one.

Proving results on rule-based systems, such as soundness or completeness, makes heavy use of advanced logic constructs, thus typically involves complex reasoning steps. Proof assistants, such as Coq [2] and Isabelle/HOL [17], have been successfully used for this task during years. It has been recently shown [8] that also automatic program verification environments are suitable for formalization of rule-based systems. Dafny is a program verifier developed since 2010 [13] that includes a programming language and specification constructs. The Dafny user creates and verifies both specifications and implementations. Dafny specification language extends first-order logic with algebraic data types, extreme/inductive predicates, induction (also co-induction), generic types, abstracting and refining modules, assertions and many others built-in specification features that makes Dafny a good candidate for our experiment. Among the specification constructs, *inductive predicates and inductive lemmas* are crucial to encode inference rules and to prove properties of the inference system, respectively. They are present in proof assistants [2, 17] and also automatic verification environments, in particular in Why [3] and Dafny [16]. Dafny is powered by the SMT solver Z3 that is a fully automatic theorem prover. Assertions are sent to Z3 to check its satisfiability that will be reported to the Dafny user. Dafny is not the only software verification tool where this kind of formalization can be done. For instance, ACL2 [12], VCC [9], VeriFast [11] or Why [10] are also suitable and powerful tools. A rule-based system formalization in a program verifier provides also a good basis to construct verified procedures implementing such systems.

In this paper, we report on the encoding of a sound and complete proof system for QCSP as a formalization into the automatic program verification environment Dafny.

⋆ Research supported by the Spanish Project FORMALISM (TIN2007-66523), by the Basque Government Project S-PE12UN050(SAI12/219), and by the University of the Basque Country under grant UFI11/45.

QCSP: Many problems can be modeled as *constraint satisfaction problems* (CSP). An instance of a CSP consists of a finite number of constraints over a fixed set of variables: the problem lies in deciding whether or not there exists an assignment to the variables which satisfies all of the constraints.

From a logic approach, the CSP problem is viewed as the problem of deciding whether existentially quantified boolean sentences are true on a given structure. This approach has proven to be very successful [6] due to the connection between the logic notion of definability and the complexity of the CSP.

Let us see some example. Recall the propositional satisfiability problem (SAT): an instance for SAT is a set of clauses such us

$$(x \vee y \vee z) \wedge (\neg x \vee \neg z \vee y) \wedge (\neg y \vee x \vee \neg z) \wedge (x \vee y \vee \neg z), \tag{1}$$

and we have to decide whether there exists a boolean assignment to variables satisfying the formula. The above example (1) is an instance of 3-SAT, because each clause contains exactly 3 literals. The variant called 2-SAT is those where the number of literals per clause is exactly 2. In general, 3-SAT is known to be NP-complete whereas that 2-SAT can be solved in polynomial time. Now, consider the QBF (the corresponding problem to SAT on quantified boolean formulas). Its instances are formulas with all variables quantified. For example,

$$\forall x\, \exists y\, \forall z((x \vee y \vee z) \wedge (\neg x \vee \neg z \vee y) \wedge (\neg y \vee x \vee \neg z) \wedge (x \vee y \vee \neg z)).$$

The problem QBF is PSPACE-complete.

We can see all these logic problems as special kinds of constraints problems. Regarding to 3-SAT, define the structure $\mathbf{A}$ with domain $\{0,1\}$ and signature $\{R_{0,3}, R_{1,3}, R_{2,3}, R_{3,3}\}$. Defines relations $R^{\mathbf{A}}_{0,3}, R^{\mathbf{A}}_{1,3}, R^{\mathbf{A}}_{2,3}, R^{\mathbf{A}}_{3,3}$ as follows.

$$R^{\mathbf{A}}_{0,3} = \{0,1\}^3 \setminus \{(0,0,0)\} \qquad R^{\mathbf{A}}_{1,3} = \{0,1\}^3 \setminus \{(1,0,0)\}$$
$$R^{\mathbf{A}}_{2,3} = \{0,1\}^3 \setminus \{(1,1,0)\} \qquad R^{\mathbf{A}}_{3,3} = \{0,1\}^3 \setminus \{(1,1,1)\}$$

For any variables, x, y, z, the relation $R^{\mathbf{A}}_{0,3}$ contains exactly all the assignments which satisfy the clause $(x \vee y \vee z)$; the relation $R^{\mathbf{A}}_{1,3}$ contains those which satisfy the clause $(\neg x \vee y \vee z)$; the relation $R^{\mathbf{A}}_{2,3}$ contains the assignments which satisfy the clause $(\neg x \vee \neg y \vee z)$; and the relation $R^{\mathbf{A}}_{3,3}$ contains those which satisfy the clause $(\neg x \vee \neg y \vee \neg z)$. Every instance of 3-SAT can be viewed as an instance of CSP($\mathbf{A}$). The 3-SAT instance (1) is the CSP($\mathbf{A}$) instance over variables $\{x, y, z\}$ and constraints $\{R_{0,3}(x,y,z), R_{2,3}(x,z,y), R_{2,3}(y,z,x), R_{1,3}(z,x,y)\}$, where we should decide whether

$$\mathbf{A} \models \exists x\, \exists y\, \exists z(R_{0,3}(x,y,z) \wedge R_{2,3}(x,z,y) \wedge R_{2,3}(y,z,x) \wedge R_{1,3}(z,x,y)).$$

In general, the problem of deciding whether boolean formulas are satisfiable is equivalent to the problem of checking whether $\mathbf{A}$ is a model of some existential first-order logic[4]. In a similar way, we can define the structure $\mathbf{C}$ in such a way that CSP($\mathbf{C}$) and 2-SAT are equivalent.

The CSP is computationally intractable, as it is the particular problem CSP($\mathbf{A}$), which is NP-complete. A classical approach to understand the complexity of problems is to investigate restricted cases, with the hope of coming up the causes of the intractability as well as why some cases are tractable.

One of the most important contribution on the complexity of the CSP was presented in [18], where Schaefer showed his dichotomy theorem on boolean CSP. Such theorem classifies the complexity of CSP($\mathbf{B}$) for all possible structures $\mathbf{B}$ over a two element domain. It gives a description of the structures $\mathbf{B}$ for which CSP($\mathbf{B}$) is polynomial-time tractable, and proved that for all other structures $\mathbf{B}$, the problem CSP($\mathbf{B}$) is NP-complete. As an example of tractability we have the problem CSP($\mathbf{C}$).

This paper concerns the quantified constraint satisfaction problem (QCSP) which is a natural generalization of the CSP, known to be PSPACE-complete. QCSP can be viewed as the problem of deciding whether or not a universally and existentially quantified sentence (made with conjunctions and atoms) is true on a given structure. When such structure $\mathbf{B}$ has a two elements domain, QCSP($\mathbf{B}$) is related to the QBF problem.

[4] These formulas are made with conjunction and do not contain functions. It is said to be formulas definable in the set $\{\wedge, \exists\}$.

Specifically, we study a new proof system for QCSP introduced in [7] (denoted it by PS), which provides a way of detecting tractability cases for the general QCSP, even if the domain has more than two elements. The study of the proofs that can be generated by PS could be a good tool to discover lower bounds on the proofs size, and even on the running time of algorithms that decide about the satisfiability of formulas. In fact, PS naturally induces a notion of *consistency* for the QCSP. In the context of CSP, a consistency notion is a condition that is sufficient for the unsatisfiability of a CSP-instance and which can typically be checked by a polynomial-time algorithm. In our setting, when QCSP instances deal with sentences having *width* bounded by a constant k, checking for k*-consistency* is a uniform polynomial-time procedure that decides such QCSP instances.

However, this is not the only reason to analyze PS. Nowadays, the development of the so-called QBF-solvers faced with the challenge of generating competitive quantified proof systems. One of those is Q-resolution [5], which generalizes resolution. In Q-resolution we need to have the formulas in prenex normal form, something that PS does not require.

2 Dafny

In this section, we provide some notions of Dafny to facilitate the understanding of the paper.

Dafny [13] is an automatic program verifier that supports a mixture of programming and specification constructs. 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 –which calls to the SMT-solver Z3– in the background and provides design time feedback. Assertion violations in lemma proofs, as well as verification errors, are reported along with different informations such as the locations (of the properties) related to the error. The interested reader is referred to [15] for further information on the several ways that Dafny IDE helps to build both lemma proofs and verified software.

The basic unit of a Dafny program is the **method**. A method is a piece of executable code with a head where multiple named parameters and multiple named results are declared. Dafny has also built-in specification constructs for assertions, such as **requires** for preconditions, **ensures** for postconditions, and **assert** for inline assertions. 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. lemma is equivalent to ghost method. Using **requires** and **ensures** we specify lemmas, in general methods. The body of a lemma is its proof. Asserts provide hints to prove lemmas, in (code) methods they are inline assertions that help whenever it cannot complete a correctness proof by itself. A hint is an assertion that the verifier is required to prove. 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. For helping the user to construct proofs, in particular to guess hints, Dafny offers two features: the construct **assume** and the use of a declared (but yet not proved) **lemma**. Of course, a proof is not complete until all verification conditions have been discharged, i.e., all assume statements have been removed (or replaced by asserts), and all the lemmas have been proved. However, along the construction of a proof, we can introduce an assumed condition φ to check whether φ is the condition that Dafny needs to complete the proof. In other words, Dafny tries to complete the proof, assuming that φ is true, without having tried to prove φ. If Dafny succeeds, then "**assume** φ" should be changed to "**assert** φ" to force Dafny to prove φ. Now, if the assertion is violated, either φ is too strong a property (hence, the user should weaken it) or φ is a heavier weight property that must be separately proved. In the latter case, the user could declare a lemma (here, parameterized by one parameter x: T) like

```
lemma L(x:T)
   ensures ψ // desirable property
```

and use it to validate the **assert** φ. That is, if a concrete lemma call (namely L(a) for some a: T), placed just before "**assert** φ", works (in the sense that φ is satisfied) then the user could comment (or drop) the assert. After that, it remains to prove the lemma, i.e. to write its body. Re-usability by instantiation (of the

parameters) is an advantage of lemmas. In other words, the **assert** mechanism provides a non-instantiable lemma, which Dafny is able to prove without extra help.

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

An inductive **datatype** is declared as follows: **datatype** D⟨T⟩= ListOfCtors where ListOfCtors is the (nonempty) |-separated list of constructors for the D. Each constructor has the form: C(params) where params is a comma-delimited list of types. Optionally each of these types can be preceded by a *destructor* that returns the corresponding parameter. For example, the usual Stack type, with constructors empty and push and destructors top and pop, can be declared by: **datatype** Stack⟨T⟩= Empty | Push(top: T, pop: Stack⟨T⟩). For every constructor C, Dafny defines a discriminator C?, which is a member that returns true if and only if the datatype value has been constructed using C.

The **match** statement (resp. expression) is provided to do pattern-matching in methods (resp. functions) on values whose type is a datatype. It binds the constructor parameters to the given names and executes (resp. applies) the corresponding case. Inductive predicates allows the definition of a predicate as an extreme solution (that is, a least or greatest fixpoint) to some equation. These kind of definition is specially useful to describe the set of judgments that can be justified by a given set of inference rules, that is a proof system. Properties can be proved by induction in the construction of the (least/greatest) fixpoint of an inductive predicate P(x). These proofs involves calls to P#[_k](x), i.e. the k-iteration of P. Such properties must be coded as **inductive lemma**s (for least fixpoint) and **colemma**s (for greatest fixpoint).

Dafny built-in immutable types include: **set**, **multiset**, **map** and **seq**, which respectively denote the types of finite sets, multisets, maps, and sequences. Operations on sets and multisets include the usual operations of + (union), ∗ (intersection), and − (set difference), comparison operators <= (subset), !! (disjointness), **in** (membership), |_| (cardinality) and the multiplicity of an element x in a multiset S, which is denoted by S[x]. Similarly, for sequences, Dafny provides + (concatenation), <= (prefix), **in** (membership), |_| (length), and many other operations. The expression S[j] denotes the element at index j of sequence S.[5] Set comprehension expressions have the form

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

for defining the set of all values given by the expression E(x1,x2,...) for all finite tuples (x1,x2,...) such that P(x1,x2,...). As particular case, (**set** x : T | P(x)) is equivalent to (**set** x : T | P(x) • x).

Tuples components are indexed from zero, that is t.0 is the first component of a tuple t, and t.1, t.2, ... are the consecutive components of t.

Dafny provides a special notation that is easy to read and understand: *calculations* [14]. A calculation in Dafny is an statement that proves a property. This notation was extracted from the *calculational method* [1], whereby a theorem is established by a chain of formulas, each transformed in some way into the next. The relationship between successive formulas (for example, equality, implication, double implication, etc.) is notated, or it can be omitted if it is the default relationship (equality). In addition, the hints (usually asserts or lemma calls) that justify a step can also be notated (in curly brackets after the relationship). Calculations are written inside the environment **calc**{ }.

Like in other proof assistants (e.g. Isabelle/HOL and Coq) and verifiers (e.g. Why3 and KeY), Dafny allows proofs to be written in different styles and with different levels of description of the outcome of every logical transformation. Therefore, proof readability and easy checking by humans is part of the work of the Dafny user.

3 QCSP proof system

In this section we reproduce both the definition of the QCSP problem and the QCSP proof system, just as it was made in [7] along with the Dafny formalization.

[5] The mentioned symbols are Dafny notation. For easy reading of the code snippets, we show them as the usual mathematical symbol (e.g ∪ for union, ∈ for membership, etc.). Quantifiers use a • for separation.

QCSP instance: We assume basic familiarity with the syntax and semantics of first-order logic. A *signature* σ is a set of relation symbols; each relation symbol $R \in \sigma$ has an associated arity $\mathsf{ar}(R)$ which is an element of $\mathbb{N}$. A *structure* $\mathbf{B}$ on signature σ consists of a *domain* B of $\mathbf{B}$, which is a non-empty set and, for each symbol $R \in \sigma$, a relation $R^{\mathbf{B}} \subseteq B^{\mathsf{ar}(R)}$. When f is a mapping, we use $f[s \mapsto b]$ to denote the extension of f that maps s to b.

```
type name = char
// names for relation and variable symbols

type Signature = set⟨(name, int)⟩
// A finite set of relation symbols with associated arity

function setOfRel (S: Signature): set⟨name⟩
{
 set x | x ∈ S • x.0
}

function arity (S: Signature, s: name): int
  requires s ∈ setOfRel(S)
{
 var p :| p ∈ (set x | x ∈ S ∧ x.0 = s);
 p.1
}

datatype Structure⟨T⟩ = Str(Sig: Signature, Dom: set⟨T⟩, I: map⟨name, set⟨seq⟨T⟩⟩⟩)

predicate wfs ⟨T⟩ (B: Structure⟨T⟩)
// decides if B is a well formed (defined) structure of signature B.Sig
{
  ∀ P • P ∈ B.Sig  ⟹ (P.0 ∈ B.I  ∧ ∀ t • t ∈ B.I[P.0] ⟹ |t| = P.1 )
}
```

We define the syntax of formulas using a datatype with four constructors: for atoms, conjunction, universal and existential quantifiers. Each of these constructors has two destructors. When ϕ is a formula, we use $\mathsf{freeVar}(\phi)$ to denote the set containing the free variables of ϕ. A *quantified-conjunctive formula* (for short, *qc-formula*) is a formula over a signature built from atoms on the signature, conjunction ($\wedge$), and the two quantifiers ($\forall$, $\exists$). Note that we permit conjunction of arbitrary arity. As expected, a *qc-sentence* is a qc-formula ϕ such that $\mathsf{freeVar}(\phi) = \emptyset$. We also define functions to (partially/totally) close formulas with existential quantification.

```
datatype Formula = Atom(rel: name, par: seq⟨name⟩) |
                   And(0: Formula, 1: Formula) |
                                  Forall(x: name, A: Formula) |
                                  Exists(y: name, E: Formula)

predicate wff (S: Signature, phi: Formula) // well-formed formula
{
match phi
  case Atom(R, par) ⇒ R ∈ setOfRel(S) ∧ |par| = arity(S,R)
  case And(phi1, phi2) ⇒ wff(S, phi1) ∧ wff(S, phi2)
  case Forall(x, phi) ⇒ wff(S, phi)
  case Exists(x, phi) ⇒ wff(S, phi)
}

function freeVar(phi: Formula): set⟨name⟩
{
match phi
  case Atom(R, par) ⇒ seq2set(par)
  case And(phi1, phi2) ⇒ freeVar(phi1) + freeVar(phi2)
  case Forall(x, phi) ⇒ freeVar(phi) − {x}
  case Exists(x, phi) ⇒ freeVar(phi) − {x}
}

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

function ExistsClose(W: set⟨name⟩, alpha: Formula): Formula
  requires W ⊆ freeVar(alpha)
  ensures ∀ S • wff(S, alpha) ⟹ wff(S, ExistsClose(W, alpha))
  ensures freeVar(ExistsClose(W, alpha)) = freeVar(alpha) − W
  decreases freeVar(alpha)
```

```
{
if W = {} then alpha else var x :| x ∈ W;
                          ExistsClose(W − {x}, Exists(x, alpha))
}

function ExistsClosure(alpha: Formula): Formula
  ensures ∀ S • wff(S, alpha) ⟹ wff(S, ExistsClosure(alpha))
  ensures sentence(ExistsClosure(alpha))
{
ExistsClose(freeVar(alpha), alpha)
}
```

Let V be a set of variables. By a mapping $f : V \rightarrow B$, we mean a mapping that sends each $v \in V$ to an element $f(v) \in B$. Valuations or assignments are maps from variables to domain individual. When f is a valuation, we use $f \restriction U$ to denote its restriction to a subset U of the domain of f. We also need to extend valuations with a sequence of values to be assigned to a sequence of names (in the specified order).

```
type Valuation⟨T⟩ = map⟨name, T⟩

function projectVal⟨T⟩ (f: Valuation⟨T⟩, U: set⟨name⟩): Valuation⟨T⟩
  requires U ⊆ domain(f)
  ensures domain(projectVal(f, U)) = U
{
map s | s ∈ U • f[s]
}

function extVal⟨T⟩ (f: Valuation⟨T⟩, W: seq⟨name⟩, S: seq⟨T⟩): Valuation⟨T⟩
  requires |W| = |S|
  ensures domain(extVal(f, W, S)) = seq2set(W) + domain(f)
  ensures range(extVal(f, W, S)) ⊆ range(f) + seq2set(S)
  decreases |W|
{
if W = [] then f else extVal(f[W[0]:=S[0]], W[1..], S[1..])
}
```

The evaluation of formula φ in a structure $\mathbf{B}$ requires a valuation f of its free variables. For that reason the predicate Models has tree parameters: ϕ, the valuation f and $\mathbf{B}$. When Models(B,f,phi) is true, φ with valuation f of its free variables is true in the structure $\mathbf{B}$, and we write $\mathbf{B}, f \models \phi$. When ϕ is a sentence, f is the empty valuation, and we write $\mathbf{B} \models \phi$.

```
function apply2seq⟨U,V⟩ (f: map⟨U,V⟩, xs: seq⟨U⟩): seq⟨V⟩
  requires seq2set(xs) ⊆ domain(f)
  ensures |apply2seq(f, xs)| = |xs|
  ensures ∀ i • 0 ≤ i < |xs| ⟹ apply2seq(f, xs)[i] = f[xs[i]]
{
if xs = [] then [] else [f[xs[0]]] + apply2seq(f, xs[1..])
}

predicate Models ⟨T⟩ (B: Structure⟨T⟩, f: Valuation⟨T⟩, phi: Formula) // B, f |= phi
  requires wfs(B) ∧ wff(B.Sig, phi)
  requires range(f) ⊆ B.Dom
  requires freeVar(phi) ⊆ domain(f)
  decreases phi
{
match phi
  case Atom(R, par) ⇒ apply(f, par) ∈ B.I[R]
  case And(phi1, phi2) ⇒ Eval(phi1, f, B) ∧ Eval(phi2, f, B)
  case Forall(x, phi) ⇒ ∀ v:T • v ∈ B.Dom ⟹ Eval(phi, f[x:=v], B)
  case Exists(x, phi) ⇒ ∃ v:T • v ∈ B.Dom ∧ Eval(phi, f[x:=v], B)
}
```

We define the QCSP to be the problem of deciding, given a QCSP instance, which is a pair $(\phi, \mathbf{B})$ where ϕ is a qc-sentence and $\mathbf{B}$ is a structure that both have the same signature, whether or not $\mathbf{B} \models \phi$.

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

QCSP proof system: We present the proof system for the QCSP, which was introduced in [7]. Given a QCSP instance $(\phi, \mathbf{B})$, we see ϕ as a tree. The proof system will allow us to derive what we call *constraints*

at the various nodes of the tree. To facilitate the discussion, we will assume that each qc-sentence ϕ has, associated with it, a set I_ϕ of *indices* that contains one index for each subformula occurrence of ϕ, that is, for each node of the tree corresponding for ϕ. To easy the encoding in Dafny, indices are as sequences of 0' and 1's. Let us remark that (in general) the collection of constraints derivable at an occurrence of a subformula does not depend only on the subformula, but also on the subformula's location in the full formula ϕ. When i is an index, we use $\phi(i)$ to denote the actual subformula of the subformula occurrence corresponding to i; we will also refer to i as a *location*.

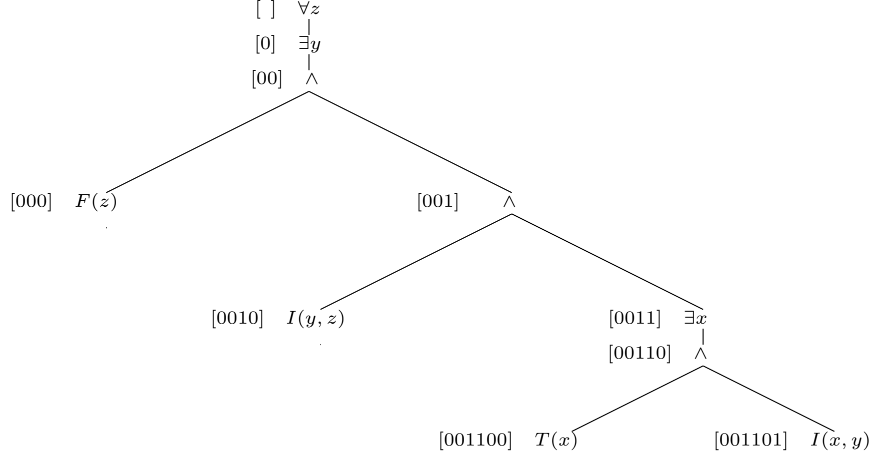

Fig. 1. Formula discussed in Examples 1 and 4

Example 1. Consider the qc-sentence $\phi = \forall z\, \exists y\, (F(z) \wedge (I(y, z) \wedge \exists x\, (T(x) \wedge I(x, y)))$. (See Figure 1.) When viewed as a tree, this formula has 10 nodes. The index set starts with the empty sequence [ ] and go on adding 0 to the left and 1 to the right. Hence, $\phi([\,]) = \forall z\, \exists y\, (F(z) \wedge (I(y, z) \wedge \exists x\, (T(x) \wedge I(x, y)))$, $\phi([0]) = \exists y\, (F(z) \wedge (I(y, z) \wedge \exists x\, (T(x) \wedge I(x, y)))$, $\phi([00]) = F(z) \wedge (I(y, z) \wedge \exists x\, (T(x) \wedge I(x, y))$, $\phi([000]) = F(z)$, $\phi([001]) = I(y, z) \wedge \exists x\, (T(x) \wedge I(x, y))$, $\phi([0010]) = I(y, z)$, $\phi([0011]) = \exists x\, (T(x) \wedge I(x, y))$, $\phi([00110]) = T(x) \wedge I(x, y)$, $\phi([001100]) = T(x)$, and $\phi([001101]) = I(x, y)$. □

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. As examples, with respect to the qc-sentence and indexing in Example 1, index [001] has two children, namely, [0010] and [0011], and index [001] has one parent, namely, [00].

The above way of indexing the abstract syntax trees of formulas is easily specified in Dafny, but we avoid to give that non-interesting details. In the sequel we explain specific references to some function/predicate related with indexed formulas.

Definition 2. 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 (U_1, F_1), (U_2, F_2) are 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 the set

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

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

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

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

The *dual-projection* will be used to eliminate universally quantified variables. Dually, *projection* can be used to cope with existential quantification.

We use 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. In Dafny we use the empty map/valuation as **map**[].

```
datatype Judgement⟨T⟩ = J(i: seq⟨int⟩,V: set⟨name⟩,F: set⟨Valuation⟨T⟩⟩)

predicate wfj⟨T⟩ (j: Judgement⟨T⟩, phi: Formula,B: Structure⟨T⟩)
// well−formed judgement (on a QCSP instance)
{
(∀ f • f ∈ j.F ⟹ (j.V = domain(f) ∧ range(f) ⊆ B.Dom))
∧
(j.i ∈ setOfIndex(phi))
∧
(j.V ⊆ freeVar(FoI(j.i, phi)))
}

predicate is_projection⟨T⟩ (j1: Judgement⟨T⟩, j2: Judgement⟨T⟩, phi: Formula,B: Structure⟨T⟩)
    // j1 is a projection of j2
  requires wfj(j1, phi,B) ∧ wfj(j2, phi,B)
{
j1.i = j2.i ∧
j1.V ⊆ j2.V ∧
j1.F = ( set f | f ∈ j2.F • projectVal(f,j1.V) )
}

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 wfj(j, phi,B) ∧ wfj(j1, phi,B) ∧ wfj(j2, phi,B)
{
j.i = j1.i = j2.i ∧
j.V = j1.V ∪ j2.V ∧
j.F = (set f | domain(f) = j1.V ∪ j2.V ∧ range(f) ⊆ B.Dom ∧ projectVal(f,j1.V) ∈ j1.F
              ∧ projectVal(f,j2.V) ∈ j2.F)
}

predicate is_dualProjection⟨T⟩ (j1: Judgement⟨T⟩,v: name, j2: Judgement⟨T⟩, phi: Formula,B: Structure⟨T⟩)
    // j1 is a dual projection of j2
  requires wfj(j1, phi,B) ∧ wfj(j2, phi,B)
{
j2.i = j1.i+[0] ∧
j1.V = j2.V \ {v} ∧
v ∈ j2.V ∧
j1.F = (set h | range(h) ⊆ B.Dom ∧ domain(h) = j1.V ∧ ∀ b • b ∈ B.Dom ⟹ h[v:=b] ∈ j2.F)
}
```

We introduce the proof system in the next definition. It is not exactly the proof system defined in [7] because we eliminate the $(\downarrow)$ rule. The authors realized that the new system (without the rule $(\downarrow)$) is also correct and complete, the proofs of its correctness and completeness are simpler, and it also enables simpler handling of k-consistency results (out of the scope of this paper).

Definition 3. A *judgement proof* on $(\phi, \mathbf{B})$ is a finite sequence of judgements, each of which has one of the following types:

(atom) $(i, \{v_1, \ldots, v_k\}, F)$
  where $\phi(i)$ is an atom $R(v_1, \ldots, v_k)$ and
  $F = \{f : \{v_1, \ldots, v_k\} \to B \mid (f(v_1), \ldots, f(v_k)) \in R^{\mathbf{B}}\}$.

(projection) $(i, U, F \restriction U)$
where (i, V, F) is a previous judgement, and $U \subseteq V$.

(join) $(i, U_1 \cup U_2, F_1 \bowtie F_2)$
where (i, U_1, F_1) and (i, U_2, F_2) are previous judgements.

($\forall$-elimination) $(i, V \setminus \{y\}, F\#(V \setminus \{y\}))$
where (j, V, F) is a previous judgement with $y \in V$,
$\phi(i) = \forall y \phi(j)$, and i is the parent of j.

(upward flow) (i, V, F)
where (j, V, F) is a previous judgement and i is the parent of j.

We say that a judgement (i, V, F) is *derivable* if there exists a judgement proof that contains the judgement. □

Let us emphasize that, by definition, in order for a triple (i, V, F) to be a judgement, it must hold that all variables in V are free variables of $\phi(i)$. Consequently, 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 j.

Next, we define the **inductive predicate** is_derivable as an encoding of the above five rules. The specification constructor **inductive predicate** has been recently introduced in Dafny ([16]) and are also called *extreme predicates*. They are boolean functions defined as an extreme solution to a recurrence relation, that is, as a least or greatest fixpoint of some functor. Such extreme predicates are specially useful when encoding a set of inductive (or coinductive) inference rules, such us the QCSP proof system above. Other tools, such as Coq [2], Isabelle/HOL [17], Agda [4], VeriFast [11], and F* [19], have since long supported inductive predicates.

```
inductive predicate is_derivable⟨T⟩ (j: Judgement⟨T⟩, phi: Formula, B: Structure⟨T⟩)
        requires wfQCSP_Instance(phi,B) ∧ wfj(j,phi,B)
{
var phii := FoI(j.i,phi);

( // by rule (atom)
  phii.Atom?
  ∧ j.V = seq2set(phii.par)
  ∧ j.F = (set f | range(f) ⊆ B.Dom ∧ domain(f) = j.V  ∧ apply2seq(f,phii.par) ∈ B.I[phii.rel])
)
∨
( // by rule (projection)
  ∃ j' • wfj(j',phi,B)
            ∧ is_projection(j,j',phi,B)
            ∧ is_derivable(j',phi,B)
)
∨
( // by rule (join)
  phii.And?
  ∧ ∃ j1,j2  • wfj(j1,phi,B) ∧ wfj(j2,phi,B)  ∧ j1.i = j2.i
                ∧ is_join(j,j1,j2,phi,B)
                ∧ is_derivable(j1,phi,B)  ∧ is_derivable(j2,phi,B)
)
∨
( // by rule (∀-elimination)
  phii.Forall?
  ∧ ∃ j' •  wfj(j',phi,B)
              ∧ phii = Forall(phii.x,FoI(j'.i,phi))
              ∧ is_dualProjection(j,phii.x,j',phi,B)
              ∧ is_derivable(j',phi,B)
)
∨
( // by rule (upward flow)
  ∃ j' • wfj(j',phi,B)
            ∧ (j'.i = j.i+[0] ∨ j'.i = j.i+[1]) ∧ j.V = j'.V
            ∧ j.F = j'.F
            ∧ is_derivable(j',phi,B)
)
}
```

Note that each of the five members of the above disjunction corresponds to the definition of one of the five rules of the proof system PS introduced in Definition 3.

Example 4. Let ϕ be the qc-sentence from Example 1 (shown in Figure 1), considered as a sentence over signature $\{T, I, F\}$ with $ar(T) = ar(F) = 1$ and $ar(I) = 2$. Define $\mathbf{B}$ to be a structure over this signature having domain $B = \{0, 1\}$ (We choose the boolean domain for a better understanding of the example): $T^{\mathbf{B}} = \{(1)\}$ (true); $F^{\mathbf{B}} = \{(0)\}$ (false); and $I^{\mathbf{B}} = \{(0, 0), (0, 1), (1, 1)\}$ (boolean implication). We may derive the next judgement proof.

1– By (atom):	$([001100], \{x\}, \{t_1\})$	where $t_1 : \{x\} \to B$ with $t_1(x) = 1$
2– By (atom):	$([001101], \{x, y\}, \{g_1, g_2, g_3\})$	$g_1(x, y) = (0, 0)$; $g_2(x, y) = (0, 1)$; $g_3(x, y) = (1, 1)$
3– By (atom):	$([0010], \{y, z\}, \{h_1, h_2, h_3\})$	$h_1(y, z) = (0, 0)$; $h_2(y, z) = (0, 1)$; $h_3(y, z) = (1, 1)$
4– By (atom):	$([000], \{z\}, \{f_1\})$	where $f_1 : \{z\} \to B$ with $f_1(z) = 0$
5– By (upward flow):	$([00110], \{x\}, \{t_1\})$	going up the judgement of 1
6– By (upward flow):	$([00110], \{x, y\}, \{g_1, g_2, g_3\})$	going up the judgement of 2
7– By (join):	$([00110], \{x, y\}, \{g_3\})$	join operator applied to judgements of 5 and 6
8– By (projection):	$([00110], \{y\}, \{(g_3 \upharpoonright \{y\})\})$	projection the judgement of 7
9– By (upward flow):	$([0011], \{y\}, \{(g_3 \upharpoonright \{y\})\})$	going up the judgement of 8
10– By (upward flow):	$([001], \{y, z\}, \{h_1, h_2, h_3\})$	going up the judgement of 3
11– By (upward flow):	$([001], \{y\}, \{(g_3 \upharpoonright \{y\})\})$	going up the judgement of 9
12– By (join):	$([001], \{y, z\}, \{h_3\})$	join operator applied to judgements of 10 and 11
13– By (projection):	$([001], \{z\}, \{(h_3 \upharpoonright \{z\})\})$	projection the judgement of 12
14– By (upward flow):	$([00], \{z\}, \{f_1\})$	going up the judgement of 4
15– By (upward flow):	$([00], \{z\}, \{(h_3 \upharpoonright \{z\})\})$	going up the judgement of 13
16– By (join):	$([00], \{z\}, \emptyset)$	join operator applied to judgements of 14 and 15
17– By (upward flow):	$([0], \{z\}, \emptyset)$	going up the judgement of 16
18– By ($\forall$-elimination:)	$([\,], \emptyset, \emptyset)$	from judgement 17.

□

4 Correctness and Completeness

The following theorem states the soundness and completeness of the previously presented proof system.

Theorem 5. Let $(\phi, \mathbf{B})$ be a QCSP instance. The empty judgement $([\,], \{\}, \{\})$ is derivable if and only if $\mathbf{B} \not\models \phi$. □

The forward direction states the correctness result that is proved in Dafny lemma correctness_Theorem. The backward direction is the completeness statement that is proved by the Dafny lemma completeness_Theorem.

For proving the correctness_Theorem we firstly prove the following Lemma 6 that is encoded in the Dafny inductive lemma correctness_Lemma on the basis of the previously defined inductive predicate is_derivable .

Lemma 6. Let $(\phi, \mathbf{B})$ be a QCSP instance and (i, V, F) a derivable judgement (on it). Let $\{v_1, v_2, \ldots v_n\}$ be the variables in $\mathsf{freeVar}(\phi(i)) \setminus V$. For all $h : V \to B$ it holds that $\mathbf{B}, h \models \exists v_1 \ldots \exists v_n \phi(i)$ implies $h \in F$. □

In Dafny notation, this lemma ensures that for any derivable judgement j, the set of valuations j.F contains any valuation h from j.V in B.Dom such that the formula

```
ExistsClose(freeVar(FoI(j.i,phi))\ j.V, FoI(j.i,phi))
```

is satisfied in the structure $\mathbf{B}$ along with the valuation h. The formula to be satisfied is obtained from the subformula of phi with index j.i (i.e. (FoI(j.i,phi))) by existentially closing the variables that are not in the domain of h (i.e. the variables in freeVar(FoI(j.i,phi))\ j.V). We define the **predicate** valuationModel to decide whether $\mathbf{B}, h \models \exists v_1 \ldots \exists v_n \phi(i)$. The **lemma** correctness_Lemma is proved on the basis of the least fixpoint of the inductive predicate is_derivable , so that this is an **inductive lemma**.

```
predicate valuationModels ⟨T⟩ (h: Valuation⟨T⟩, j: Judgement⟨T⟩, phi: Formula, B: Structure⟨T⟩)
  requires wfs(B) ∧ wff(B.Sig, phi) ∧ wfj(j, phi, B)
  requires wff(B.Sig, ExistsClose(freeVar(FoI(j.i, phi))\j.V, FoI(j.i, phi)))
{
domain(h) = j.V  ∧ range(h) ⊆ B.Dom  ∧ Models(B, h, ExistsClose(freeVar(FoI(j.i, phi))\j.V, FoI(j.i, phi)))
}

inductive lemma correctness_Lemma⟨T⟩ (j: Judgement⟨T⟩, phi: Formula, B: Structure⟨T⟩)
  requires wfQCSP_Instance(phi, B) ∧ wfj(j, phi, B)
  requires is_derivable(j, phi, B)
  ensures wff(B.Sig, ExistsClose(freeVar(FoI(j.i, phi))\j.V, FoI(j.i, phi)))
  ensures ∀ h • valuationModel(h, j, phi, B) ⟹ h ∈ j.F
⊞{...}
```

The following lemma correctness_Theorem states and proves the correctness of the proof system. We use a calculation (**calc**) that assumes the premise is_derivable (J ([], $\{\}, \{\}$),phi, B), i.e. the empty judgement J ([], $\{\}, \{\}$) is derivable in the root of the sentence phi.
The calculation in the proof of lemma correctness_Theorem consist just in one step (of implication), with two hints helping Dafny to prove the implication step. We also include three commented assertions as explanation. First, we call the correctness_Lemma applied to the empty judgement J ([], $\{\}, \{\}$) and the considered QCSP instance phi, B. Then, Dafny proves the (first commmented) assertion that ensures that any valuation with empty domain (i.e. only **map**[]) that makes the sentence phi to be true in the structure B should belong to the set of valuations of the empty judgement. Since, the set emptyj.F is empty, we get there is no evaluation h belonging to the set of valuations that makes B to be a model of phi. Hence, second, we assert that, in particular, the empty map **map**[] does not belong to the set of valuations that makes B to be a model of phi. Hence, with these two hints, ¬Models(B,**map**[],phi) (i.e. $B \not\models \varphi$) is proved from the premise is_derivable (J ([], $\{\}, \{\}$),phi, B).

```
lemma correctness_Theorem⟨T⟩ (phi: Formula, B: Structure⟨T⟩)
  requires wfQCSP_Instance(phi, B)
  requires is_derivable(J([], {}, {}), phi, B)
  ensures ¬Models(B, map[], phi)
{
var emptyj: Judgement⟨T⟩ := J([], {}, {});
calc {
     is_derivable(emptyj, phi, B);
     ⟹ {
        correctness_Lemma(emptyj, phi, B);
        // assert ∀ h • valuationModel(h, emptyj, phi, B) ⟹ h ∈ emptyj.F;
        // assert emptyj.F = {};
        // assert ∀ h • ¬valuationModel(h, emptyj, phi, B);
        assert ¬valuationModel(map[], emptyj, phi, B);
        }
     ¬Models(B, map[], phi);
     }
}
```

The Dafny lemma completeness_Theorem (i.e the backward direction of Lemma 5), is proved on the basis of the following Lemma 7 which is encoded in the Dafny lemma canonical_judgement_Lemma. For that, we define a function canonical_judgement is on charge of computing the existing judgement $(i, \mathsf{freeVar}(\phi(i)), F)$ provided by Lemma 7.

Lemma 7. Let $(\phi, \mathbf{B})$ be a QCSP instance. Let I_ϕ be the index set of ϕ. For each $i \in I_\phi$, there exists, at least, a derivable judgement $(i, \mathsf{freeVar}(\phi(i)), F)$ where $h \in F$ if and only if $\mathbf{B}, h \models \phi(i)$. □

Hence, in order to prove completeness, we write a constructive proof that firstly associate the above judgement $(i, \mathsf{freeVar}(\phi(i)), F)$ to each index i of the formula whose variables are exactly the free variables of the subformula of phi with index i. We call it the *canonical judgement*. It is recursively defined by the following function canonical_judgement whose well-foundedness is given by the decreasing expression FoI(i, phi). The parameters of this function are not only i and phi, but also the structure B that is required to obtain the set F of valuations of the judgement. Note that the function canonical_judgement has four ensures that are automatically proved. Note also that, before the match expression, there are two lemma calls: setOfIndexSuffix_Lemma(i, phi); and indexSubformula_Lemma (i,phi);. These two lemma calls, respectively, prove

that the indexes i+[0] and i+[1] occurring in the recursive calls to canonical_judgement are indexes of some subformula of phi and they are the indexes of the subformulas named in the corresponding pattern.

```
function canonical_judgement⟨T⟩ (i: seq⟨int⟩, phi: Formula, B: Structure⟨T⟩): Judgement⟨T⟩
  requires wfQCSP_Instance(phi,B)
  requires i ∈ setOfIndex(phi)
  ensures canonical_judgement(i,phi,B).i = i
  ensures canonical_judgement(i,phi,B).V = freeVar(FoI(i,phi))
  ensures wfj(canonical_judgement(i,phi,B),phi,B)
  ensures ∀ f • f ∈ canonical_judgement(i,phi,B).F ⟹ (domain(f) = freeVar(FoI(i,phi))
                                                          ∧ range(f) ⊆ B.Dom)
  decreases FoI(i,phi)
{
var phii := FoI(i,phi);
setOfIndexSuffix_Lemma(i,phi);
indexSubformula_Lemma (i,phi);
match phii
      case Atom(R,par) ⇒ var F := (set f | range(f) ⊆ B.Dom ∧ domain(f) = seq2set(par)
                                        ∧ apply2seq(f,par) ∈ B.I[R]);
                          J(i, seq2set(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 ∈ 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 j0' := projection(j0,j0.V\{x},phi,B);
                            J(i,j0'.V,j0'.F)
}
```

In the above definition, we have make use of the following three functions for defining the projection , the join , and the dual_projection of a judgement:

```
function projection⟨T⟩ (j: Judgement⟨T⟩,U: set⟨name⟩, phi: Formula,B: Structure⟨T⟩): Judgement⟨T⟩
  requires wfj(j,phi,B)
  requires U ⊆ j.V
  ensures wfj(projection(j,U,phi,B),phi,B)
  ensures is_projection(projection(j,U,phi,B),j,phi,B)
{
J(j.i,U,(set f | f ∈ j.F • projectVal(f,U)))
}

function join⟨T⟩ (j1: Judgement⟨T⟩, j2: Judgement⟨T⟩, phi: Formula,B: Structure⟨T⟩): Judgement⟨T⟩
  requires wfj(j1,phi,B) ∧ wfj(j2,phi,B)
  requires j1.i = j2.i
  ensures wfj(join(j1,j2,phi,B),phi,B)
  ensures is_join(join(j1,j2,phi,B),j1,j2,phi,B)
{
J(j1.i,j1.V ∪ j2.V,(set f | domain(f) = j1.V+j2.V ∧ range(f) ⊆ B.Dom
                          ∧ projectVal(f,j1.V) ∈ j1.F
                          ∧ projectVal(f,j2.V) ∈ j2.F))
}

function dualProjection⟨T⟩ (v: name, j: Judgement⟨T⟩, phi: Formula,B: Structure⟨T⟩): Judgement⟨T⟩
  requires wfj(j,phi,B)
  requires |j.i| ⟩ 0 ∧ j.i = j.i[..|j.i|−1] + [0]
  requires j.i[..|j.i|−1] ∈ setOfIndex(phi)
  requires FoI(j.i[..|j.i|−1],phi).Forall? ∧ FoI(j.i[..|j.i|−1],phi).x = v
  requires v ∈ j.V
  ensures wfj(dualProjection(v,j,phi,B),phi,B)
  ensures is_dualProjection(dualProjection(v,j,phi,B),v,j,phi,B)
{
indexSubformula_Lemma(j.i[..|j.i|−1],phi);
J(j.i[..|j.i|−1], j.V\{v}, (set f | range(f) ⊆ B.Dom ∧ domain(f) = j.V\{v}
                                   ∧ ∀ b • b ∈ B.Dom ⟹ f[v:=b] ∈ j.F))
}
```

It is worthy to note that the previously defined three predicates is_projection , is_join , and is_dual_projection could be respectively defined in terms of the three functions projection , join , and dual_projection . In fact, Dafny automatically checks that the judgement returned by each function satisfies the corresponding predicate. Since this correspondence is ensured, we decided to keep the previous definitions of the three predicates with

the certainty that there are not incoherences. Note also that the well-formedness of the returned judgement is also automatically ensured.

The following lemma canonical_judgement_Lemma ensures that the canonical judgement is derivable and contains exactly all valuations in the set setOfValModels(Fol(i, phi), B. In other words, each subformula Fol(i, phi) of index i is satisfied in the structure exactly for the valuations f belonging to the canonical judgement for index i.

```
lemma canonical_judgement_Lemma⟨T⟩ (i: seq⟨int⟩, phi: Formula, B: Structure⟨T⟩)
  requires wfQCSP_Instance(phi,B)
  requires i ∈ setOfIndex(phi)
  ensures canonical_judgement(i, phi, B).i = i
  ensures canonical_judgement(i, phi, B).V = freeVar(Fol(i, phi))
  ensures canonical_judgement(i, phi, B).F = setOfValModels(Fol(i, phi),B);
  ensures is_derivable(canonical_judgement(i, phi, B), phi, B)
  decreases Fol(i, phi)
⊞{...}
```

The completeness_Theorem proves that whenever the sentence phi is not satisfied by the structure B, then the empty judgement of index [] is derivable. In the proof we start calling the canonical_judgement_Lemma with index []. Commented assertions are included as documentation, Dafny is able to prove them automatically. However, the assertion canonical_judgement ([], phi, B).F = {} is not automatically proved. We include a calculation that proves, by *reductio ad absurdum*, that the set of valuations belonging to the canonical judgement of the root of phi is empty.

```
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)
{
canonical_judgement_Lemma([], phi, B);
// assert canonical_judgement([], phi, B).i = [];
// assert canonical_judgement([], phi, B).V = {};
∀ f: Valuation⟨T⟩
     ensures f ∉ canonical_judgement([], phi, B).F;
     { calc {
            f ∈ canonical_judgement([], phi, B).F;
            ⟹
            range(f) ⊆ B.Dom ∧ domain(f) = {} ∧ Models(B, f, phi);
            ⟹
            range(f) ⊆ B.Dom ∧ f = map[] ∧ Models(B, f, phi);
            ⟹
            range(f) ⊆ B.Dom ∧ Models(B, map[], phi);
            ⟹
            false;
            }
     }
assert canonical_judgement([], phi, B).F = {};
// assert is_derivable(J([],{},{}), phi, B);
}
```

5 Experience and Further Work

We have encoded a QCSP proof system in a reasonably natural and intuitive way, using an inductive predicate of derivability whose meaning is given by the least fixpoint semantics. The interaction with the tool (Dafny) has been agile and fast. For checking details, Dafny provided useful information about what is not yet correct and why. We have realized that the SMT solver behind Dafny supports a number of proof features traditionally found in only interactive proof assistants like Coq and Isabelle. This task is been very interesting from the point of view of given a detailed formalization and for debugging the proofs which were previously written with pen and paper in a more imprecise form. Also remark that automated proofs often require proving some essential properties that are usually assumed (in the concerned area) without any proof. In our formalization, this is the case with a few lemmas about logical equivalences between syntactically different existential closures of formulas.

The formalization of the proof system consists of four modules Utils, QCSP_Instance, Correct_Proof_System, PS_Completeness. Each module is a file .dfy that imports the previous ones. We estimate that the human

effort required for the current encoding is between 120 and 150 hours, though it is not already the final version, some easy lemmas remain to be proved. Moreover, we plan to improve the first complete version for easy reading and also for computational performance. The successive versions of this formalization (and its future extensions) will be available through `http://www.sc.ehu.es/jiwlucap/QCSPinDafny.html`. Next, we try to give the reader a rough idea of the size and computational cost of the current formalization. We report on the seconds required to verify each module by Dafny 1.9.9.40414 for Windows(x64) running by a processor i7-6600U CPU at 2.60GHz 2.80GHz with 16 GB of RAM. The module Utils contains basic definitions and lemmas –about sequences, sets and maps– that are useful for the other three modules. This first module consists of 5 functions and 5 lemmas on about 125 lines.[6] Its verification takes about 3,5 seconds. The module QCSP_Instance defines what is a well-formed QSCP-instance, where formulas and structures are involved. Then, we define many related notions, such as model, valuation, project, extension of a valuation and the existential closure of a formula. Finally, many properties relating models with transformation of valuations and with existential closure are proved. Along about 500 lines, we essentially define 2 datatypes, 5 predicates, 8 functions and 16 lemmas. The larger proof (of lemma ExistsCloseSum_Lemma) is about 30 lines and includes 4 calls to other lemmas and 2 calls to itself (induction hypothesis). The whole module takes about 30 seconds. The proof of ExistsCloseSum_Lemma takes about 2,5 seconds. for solving 76 proof obligations. In the module Correct_Proof_System we define the proof system and prove its correctness using about 400 lines. It consists of 2 datatypes, 4 functions, 8 predicates (two of then inductive) and 11 lemmas (one of them inductive). The whole module is verified in about 27 seconds and the proof of the inductive lemma correctness_Lemma checks 158 proof obligations in about 2,7 seconds. The proof of completeness, in module PS_Completeness, have about 200 lines defining 5 functions and proving 2 lemmas. It is verified in 10,5 seconds. The poorf of the lemma canonical_judgement_Lemma generates 182 proof obligations that are verified in about 4 seconds.

As future work, we are particularly interested in developing automatic proofs for the tractability results derived from the proof system. The proof of these results are tedious, repetitive, and tricky. It is easy to make mistakes in hidden details. It is in these kinds of situations where Dafny helps to convince the user about the correctness of the proofs. We aim to provide valuable assistance to researchers who try to identify polynomial-time restricted versions of the QCSP problem. This could require to extend the introduced proof system to a broader language, for that we plan to reuse the presented formalization by extending it.

References

1. Roland Backhouse. The calculational method. *Information Processing Letters*, 53(3):121, 1995.
2. Yves Bertot, Pierre Castran, Grard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Donnes complmentaires http://coq.inria.fr.
3. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.
4. Ana Bove, Peter Dybjer, and Ulf Norell. 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, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.
5. H.K. Buning, M. Karpinski, and A. Flogel. Resolution for quantified boolean formulas. *Information and Computation*, 117(1):12 – 18, 1995.
6. Hubie Chen. A rendezvous of logic, complexity, and algebra. *ACM Comput. Surv.*, 42(1):2:1–2:32, 2009.
7. Hubie Chen. Beyond q-resolution and prenex form: A proof system for quantified constraint satisfaction. *Logical Methods in Computer Science*, 10(4), 2014.
8. Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. *Formalizing Semantics with an Automatic Program Verifier*, pages 37–51. Springer International Publishing, Cham, 2014.
9. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. *VCC: A Practical System for Verifying Concurrent C*, pages 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[6] Including white and commented lines (here and in the sequel).

10. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems — 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
11. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frdric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In M.G. Bobaru, K. Havelund, G.J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, 2011.
12. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-aided reasoning : an approach*. Advances in formal methods. Kluwer Academic Publishers, Boston, 2000.
13. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355, pages 348–370. Springer, April 2010.
14. K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers*, volume 8164, pages 170–190. Springer, 2014.
15. K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014*, volume 149 of *EPTCS*, pages 3–15, April 2014.
16. Rustan Leino. Well-founded functions and extreme predicates in dafny: A tutorial. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*, volume 40 of *EPiC Series in Computing*, pages 52–66. EasyChair, 2016.
17. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
18. Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM.
19. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.