qRHL tool – Manual

Dominique Unruh
University of Tartu

This is a user manual for our proof assistant for performing qRHL-based security proofs. The tool is a prototype to demonstrate the logic and to experiment with security proofs. At this point, it is not yet meant for larger developments.

This manual assumes knowledge of the underlying qRHL formalism, see [9].

The source code is published on GitHub [8].

For installation instructions see the included `README.md`.

1 Architecture

The tool consists of three main components: a ProofGeneral [6] frontend, the core tool written in Scala, and a Isabelle/HOL [4] backend with custom theories. The ProofGeneral frontend merely eases the interactive development of proofs, once a proof script is finished, it can also be checked by the core tool directly. The core tool implements a theorem prover for qRHL (with tactics-based backward reasoning). Only tactics for manipulating qRHL judgements are built-in into the core tool. Many tactics produce subgoals that are not qRHL judgments. (We call those *"ambient" subgoals* because they are expressed in the ambient logic.) Those ambient subgoals are outsourced to the Isabelle/HOL backend for simplification or solving. This way, the overall tool supports arbitrarily complex pre- and post-conditions in qRHL statements, and arbitrarily complex expressions within programs (only limited by what can be expressed in Isabelle/HOL). The Isabelle/HOL backend is automatically downloaded, compiled, and executed by the core tool (via libisabelle [3]).[1]

More precisely, when parsing a program, all expressions (e.g., `1+2` in an assignment `a <- 1+2`) are sent as literal strings to Isabelle/HOL for parsing. And in a qRHL judgement such as `{𝔈𝔩𝔞[x1=x2]} x <- x+1; ~ skip; {𝔈𝔩𝔞[x1≠x2]}`, the predicates `𝔈𝔩𝔞[x1=x2]` and `𝔈𝔩𝔞[x1≠x2]` are also parsed by Isabelle/HOL. In order to support the different constructions used in predicates (see Section 4 in [9], e.g., $\mathfrak{Ela}[\ldots]$ or $\equiv_{\mathsf{quant}}$), we include an Isabelle/HOL theory `QRHL.thy` in the tool that contains the definitions and simplification rules needed for reasoning about quantum predicates.

We stress that although we use Isabelle/HOL as a backend, this does not mean that our tool is an LCF-style theorem prover (i.e., one that breaks down all proofs to elementary mathematical proof steps). All tactics in the tool, and many of the simplification rules in `QRHL.thy` are axiomatized (and backed by the proofs in this paper).[2] We simply use Isabelle/HOL as a backend because it comes with rich existing theories and tools. Embedding it in our tool avoids duplication of effort.

A proof script for our tool consists of a UTF-8 encoded qRHL file `example.qrhl`, optionally accompanied by an Isabelle/HOL theory `Example.thy`. See Figure 1. The accompanying Isabelle/HOL theory can define additional constants (e.g., `square`) and simplification rules (e.g., `square_simp`), etc. All files (including user-created ones) are expected to be found in top-level directory of the tool's installation directory.

To execute the example, execute `./proofgeneral.sh example.qrhl` and then use, e.g., `Ctrl-C Ctrl-N` to evaluate the file step by step. (If emacs is not available, you can also run `bin/qrhl example.qrhl` noninteractively.) To edit `Example.thy`, execute `./run-isabelle.sh Example.thy`.

[1] This uses about 2GB of additional disk space. The downloaded Isabelle is stored in a subdirectory of the tools installation, so deleting the tools directory will also recover that space.

[2] The theory `QRHL.thy` is integrated in executable in the binary distribution but can be inspected at `https://github.com/dominique-unruh/qrhl-tool/blob/master/src/main/isabelle/QRHL.thy`.

example.qrhl

```
isabelle Example.

classical var c : nat.
quantum var q : bit.

program P1 := { c <- square 2; }.

qrhl {Cla[c2 = 4]} call P1; ~ skip;
     {Cla[c1 = c2]}.
  inline P1.
  wp left.
  skip.
  simp!.
qed.

lemma test: 1+1=2.
  simp!.
qed.
```

Example.thy

```
theory Example
  imports QRHL
begin

definition "square x = x*x"

lemma square_simp[simp]:
  "square x = x*x"
  using square_def by auto

end
```

Figure 1: Example qRHL proof script. The files are bundled with the tool.

2 Proof scripts

qRHL proof scripts contain a mixture of declarations (e.g., defining a variable or a program), claims (e.g., qRHL judgements), and proofs. Syntactically, the script is a sequence of commands.

A command is a single or multiline string, terminated with a "`.`". (The "`.`" must be the last non-whitespace character in the line. That is, no further commands or comments can be in the same line.) Inside a command, line breaks are treated like spaces.

Between commands, there can be comments that start with "`#`" (on their own lines). These are ignored. (Comments may not occur within a command, not even inside a multiline command.)

Isabelle initialization. The first command in a proof script must be "`isabelle.`" This initializes (and possibly downloads, if needed) Isabelle/HOL. If a custom Isabelle/HOL theory "`Example.thy`" is to be used, use the command "`isabelle Example.`" instead. Custom Isabelle/HOL theories should import the theory `QRHL` to get access to qRHL-related definitions, lemmas, and simplification rules. See Section 6 for more information on accompanying theories.

Declaring variables. There are three different kinds of variables: classical, quantum, and ambient variables. Classical and quantum variables represent classical and quantum program variables as defined in [9]. These can be declared using the following commands

```
classical var x : type.
quantum var q : type.
```

respectively. Here `x`,`q` are the variable names, and `type` is the type of the variable. That is, $\mathsf{Type}_{\mathtt{x}} = \mathrm{UNIV}_{\mathtt{type}}$ where $\mathrm{UNIV}_{\mathtt{type}}$ is the universe of all values of type `type`. `type` can be an arbitrary type that is understood by Isabelle/HOL. (If custom types are needed, they can be defined in an accompanying theory. Simple examples of predefined types are `bit`, `bool`, `nat` (natural numbers), `int` (integers).) Any program variable that is used anywhere in the proof script must be declared. If a variable `x` was defined, then the names `x1` and `x2` are available in predicates to refer to that variable in the left/right program.

An ambient variable simply stands for a fixed but arbitrary value. That is, ambient variables are implicitly all-quantified. In other words, ambient variables are free variables of the ambient logic. Ambient variables are declared using

```
ambient var x : type.
```

where `type` is again an arbitrary Isabelle/HOL type.

Program declarations. There are two kinds of declarations for programs. The first is

```
program name := { code }.
```

which defines `name` to refer to the program described by `code`. Logically, this simply introduces an abbreviation for referring to a concrete code fragment. This code fragment can then be embedded in other code fragments (see the `call` statement in the syntax of programs, Section 3). For the syntax of `code`, see Section 3.

The second kind of declaration declares an unspecified program:

```
adversary name vars v1,v2,v3,...,vn.
```

That is, after this declaration, `name` is assumed to refer to some program containing (at most) the program variables `v1`,...,`vn`. Nothing beyond that restriction on its variables is assumed. Thus, if we prove a statement referring to `name`, this statement holds for any program `name`. We use these declarations to model adversaries.

Since the language in this paper does not model procedure calls, adversaries are simply program fragments that get executed as part of a larger program. In particular, there is no syntactic provision for inputs and outputs of an adversary. Instead, all communication with the adversary has to take place through global variables. We recommend the following approach to the definition of adversaries: One declares two variables for the internal state of the adversary (one classical, one quantum), declares some variables for input/output of the adversary (as needed in the specific context where the adversary is used), and then declares an adversary that uses all those variables (with an informal comment detailing which variables are intended as input and output). For example, in `prg-enc-rorcpa.qrhl` (see Section 7.1), we have an adversary `A2` that takes an message `c` and returns a bit `b`. The declaration is:

```
quantum var qglobA : string.
classical var cglobA : string.
classical var c : msg.
classical var b : bit.
# A2: inputs: c; outputs: b
adversary A2 vars c,b,cglobA,qglobA.
```

(Here the adversary state is in `cglobA` and `qglobA`, those variables are also shared with other program fragments representing different invocations of the same adversary. We use the type `string` for the state to ensure that the type is big enough to allow to represent any computation.)

Note that this approach also allows us to model adversaries that cannot communicate by simply giving them no shared global variables.

Furthermore, in a reduction-based security proof, we need to construct a new adversary `B` from an existing adversary `A`. This can be done by using the `program` to define a new adversary `B` that invokes the existing (unspecified) adversary program `A`. For example, `prg-enc-rorcpa.qrhl` defines:

```
# B: inputs: r; outputs: b
program B := { call A1; c <- r+m; call A2; }.
```

Goals. To start a proof, one first needs to state a goal. There are two kinds of goals: qRHL judgements, and ambient logic statements. A qRHL judgement goal is opened using the `qrhl`-command:

```
qrhl {pre} code1 ~ code2 {post}.
```

Here `pre` and `post` are quantum predicates (parsed by Isabelle/HOL, see Section 4), and `code1`, `code2` are programs (see Section 3 for the syntax). The meaning of this command is that we start a proof of the qRHL judgment {`pre`}`code1` ~ `code2`{`post`}.

The second kind of goal is an ambient logic goal, opened using the `lemma`-command:

```
lemma name: formula.
```

Here `name` is the name under which the proven fact will be stored. And `formula` is an arbitrary formula that Isabelle/HOL understands. For example,

```
lemma test: 1+1=2.
```

starts the proof of a lemma called `test` of the fact that $1+1=2$. Once a lemma is proven, the new fact can be referred to like any other fact known to Isabelle/HOL, for example when using the tactic `simp`.

Note that `formula` cannot contain qRHL judgments. (These have no encoding in Isabelle/HOL.) It is, however, possible to refer to named programs (declared using the `program`-command or the `adversary`-command) in Isabelle/HOL expressions of the following form:

```
Pr[b:prog(rho)]
```

Here `b` must be an expression of type `bool`, and `prog` must be the name of a declared program, and `rho` must be an expression of type `program_state` (typically `rho` is simply an uninterpreted ambient variable). Then `Pr[b:prog(rho)]` denotes the probability that `b` = `true` after executing `prog` with initial state ρ (as in Definition 9 in [9]). For example,

```
lemma secure: Pr[b=1:game1(rho)] = Pr[b=1:game2(rho)].
```

would start a goal stating that the programs `game1` and `game2` have the same probability of outputting 1 in variable `b`, for any initial state. Such goals can be transformed into qRHL goals using the tactic `byqrhl`, but they can also be reasoned about in Isabelle/HOL (via the `simp` tactic) which treats those probabilities as uninterpreted values $\in [0,1]$.

Proofs. Once a goal has been opened using either `qrhl` or `lemma`, the tool is in proof mode. In this mode, the state consists of a list of subgoals. (In ProofGeneral, the current list of subgoals are listed in the `*goals*` window.) Each subgoal is either a qRHL judgment (like the ones created by the `qrhl` command) or an ambient logic formula (like the ones created by `lemma`). (A qRHL subgoal can additionally contain a list of assumptions $A_1, \ldots, A_n$ that are ambient logic formulas. In this case, the interpretation is that the qRHL judgments holds whenever those assumptions are satisfied.)

A proof consists of a sequence of tactic invocations. Each tactic transforms the first subgoal into zero or more subgoals. (With the guarantee that the new goals together imply the original subgoal.) The available tactics are described in Section 5 below.

When the list of subgoals is empty, the proof must be finished by

```
qed.
```

This finishes the proof, and further declarations can be made, or new goals opened. If the current proof started with a `lemma` command, the proven fact is stored under the name specified in the `lemma` command.

3 Programs

A program is represented as a list of statements.[3] Each statement is one of the following:

Syntax	**Meaning**
`skip;`	The empty program **skip**.
x `<-` *expr*`;`	The assignment $x \leftarrow expr$. x must be declared as a classical variable of some Isabelle/HOL type T, and *expr* must be an Isabelle/HOL term of the same type T. *expr* may contain classical and ambient variables as free variables. **Example:** "`x <- x+1;`" increases `x`.

[3] This deviates slightly from the syntax of programs described in Section 3.2 in [9]. There, larger programs are composed from smaller ones by using the binary sequential composition operation ";". However, since the sequential composition is associative (up to denotational equivalence), we can instead represent a nested application of sequential compositions as a simple list of statements.

x `<$` *expr*`;`	The sampling $x \xleftarrow{\$} expr$. x must be declared as a classical variable of some Isabelle/HOL type T, and *expr* must be an Isabelle/HOL term of the type T `distr`, the type of distributions over T. (See the tables below for constants for constructing distributions.) *expr* may contain classical and ambient variables as free variables. **Example:** "`x <$ uniform UNIV;`" samples `x` uniformly from the type of `x` (assuming the type of `x` is finite).
q_1`,...,`q_n `<q` *expr*`;`	The quantum initialization $q_1, \ldots, q_n \xleftarrow{\mathbf{q}} expr$. $q_1, \ldots, q_n$ must be declared as quantum variables with some Isabelle/HOL types $T_1, \ldots, T_n$. All q_i must be distinct variables. *expr* must be an Isabelle/HOL expression of type $(T_1 \times \cdots \times T_n)$ `vector`, the type of vectors with basis $T_1 \times \cdots \times T_n$. (See the tables below for constants for constructing states.) *expr* may contain classical and ambient variables as free variables. Note that our definition of well-typed programs (Section 3.2 in [9]) requires *expr* to be a unit vector, while in our tool, we allow *expr* to be a non-normalized vector. This is simply to avoid having to define too many different types in Isabelle/HOL (which would lead to the need of applying type conversions very often). The tactics in the tool take this into account and create an explicit precondition that *expr* has unit length (specifically, the tactic `wp` which implements rule QInit1).[4] **Example:** "`x,y <q EPR;`" initializes `x`,`y` to contain an EPR pair. (Assuming that `x` and `y` are quantum variables of type `bit`.)
x `<- measure` q_1`,...,`q_n `with` *measurement*`;`	The measurement $$x \leftarrow \textbf{measure } q_1, \ldots, q_n \textbf{ with } measurement$$ x must be declared as a classical variable of some Isabelle/HOL type T_x. $q_1, \ldots, q_n$ must be declared as quantum variables of some Isabelle/HOL types $T_1, \ldots, T_n$. All q_i must be distinct variables. *measurement* must be an Isabelle/HOL expression of type $(T_x, T_1 \times \cdots \times T_n)$ `measurement`, the type of measurements with outcomes of type T_x. (See the tables below for constants for constructing measurements.) *expr* may contain classical and ambient variables as free variables. **Example:** "`x <- measure q with computational_basis;`" measures the quantum variable `q` in the computational basis and assigns the outcome to the classical variable `x`. Both variables must have the same type.

[4] Formally, changing the type of programs is justified as follows: A program q_1`,...,`q_n `<q` *expr*`;` is interpreted as $q_1, \ldots, q_n \xleftarrow{\mathbf{q}} mkUnit(e)$ where $mkUnit(\psi) = \psi$ for unit vectors ψ, and $mkUnit(\psi)$ is an arbitrary unit vector if ψ is not a unit vector. With this interpretation, programs as implemented in our tool match the typing-rules and semantics in [9]. See footnote 9 for how this affects the rules implemented by the tactics.

`on` q_1`,...,`q_n `apply` *expr*`;`	The unitary quantum operation **apply** *expr* **to** $q_1, \ldots, q_n$. $q_1, \ldots, q_n$ must be declared as quantum variables of some Isabelle/HOL types $T_1, \ldots, T_n$. All q_i must be distinct variables. *expr* must be an Isabelle/HOL expression of type $(T_1 \times \cdots \times T_n, T_1 \times \cdots \times T_n)$ `bounded`, the type of bounded operators. (See the tables below for constants for constructing bounded operators.) *expr* may contain classical and ambient variables as free variables. Note that our definition of well-typed programs (Section 3.2 in [9]) requires *expr* to be an isometry, while in our tool, we allow *expr* to be any bounded operator. This is simply to avoid having to define too many different types in Isabelle/HOL (which would lead to the need of applying type conversions very often). The tactics in the tool take this into account and create an explicit precondition that *expr* is an isometry (specifically, the tactic `wp` which implements rule QApply1).[5] **Example:** "`on x,y apply CNOT;`" applies a CNOT to the quantum variables `x`,`y`. (They are assumed to be of type `bit`.)
`if (`c`) then` P_1 `else` P_2	The conditional **if** c **then** P_1 **else** P_2. c must be an Isabelle/HOL expression of type `bool`. c may contain classical and ambient variables as free variables. The programs P_1 and P_2 are either single statements, or blocks of the form `{` s_1 s_2 `...` s_n `}` where each s_i is a statement. (Note that each s_i will end with a semicolon.) **Example:** "`if (x=0) then x <- x+1; else skip;`" is equivalent to `x <- 1;` (assuming `x` is of type `bit`). **Example:** "`if (x=0) then { x <- 1; y <- 1; } else { x <- 0; y <- 0; }`" sets `x` and `y` to 1 if `x=0`, and to `0` otherwise.
`while (`c`) then` P	The conditional **while** c **do** P. c must be an Isabelle/HOL expression of type `bool`. c may contain classical and ambient variables as free variables. The programs P is either a single statement, or a block of the form `{` s_1 s_2 `...` s_n `}` where each s_i is a statement. (Note that each s_i will end with a semicolon.) **Example:** "`while (x≤0) x <- x+1;`" increases `x` until it is positive (assuming `x` is of type `int`). **Example:** "`while (x≤0) { x <- x+1; y <- y+1; }`" increases both `x` and `y` until `x` is positive.

[5]Formally, changing the type of programs is justified as follows: A program `on` $q_1, \ldots, q_n$ `apply` e`;` is interpreted as **apply** $mkIso(e)$ **to** $q_1, \ldots, q_n$ where $mkIso(U) = U$ for isometries U, and $mkIso(U)$ is an arbitrary isometry (e.g., the identity) if U is not an isometry. With this interpretation, programs as implemented in our tool match the typing-rules and semantics in [9]. See footnote 8 for how this affects the rules implemented by the tactics.

`call` *prog*`;`	The program *prog* itself. *prog* must be the name of a program (declared with `program` or `adversary`). Logically, `call` *prog*`;` is simply an abbreviation for the code of *prog*. And if *prog* was defined using `program`, it would be equivalent to simply write the code from the definition of the program instead of `call` *prog*`;`. (Although some tactics may treat the two cases differently.) However, if *prog* was defined using `adversary`, the `call` *prog*`;` syntax is necessary since the code of *prog* is not known. We do not have a corresponding construct in the syntax from Section 3.2 in [9] because we can simply write *prog* instead of `call` *prog*`;`. (For example, `x <- 1; call A; x <- 0;` translates to $x \leftarrow 1; A; x \leftarrow 0$.) Note that `call` is not a procedure call. In particular, we cannot pass arguments, have local variables, or get a return value. However, arguments and return values can be emulated by using global variables (see the discussion of program declarations in Section 2). **Example:** "`call A;`" invokes the adversary `A` (assuming `A` was declared using `adversary A vars ...`).

4 Expressions and predicates

Expressions. Expressions within programs, and predicates in qRHL judgments are interpreted by Isabelle/HOL (the Isabelle/HOL 2017 version), in the context of a builtin theory QRHL. We assume some familiarity with Isabelle/HOL. Readers unfamiliar with Isabelle/HOL may study the tutorial [5].

For experiments, it can be useful to directly invoke Isabelle/HOL (using the `./run-isabelle.sh` script) and edit a theory that imports `QRHL`.

Expressions used in assign-statements will probably only rarely use any of the custom types and constants from `QRHL.thy` (except the type `bit`). However, in sampling-statements we need to construct expressions of type α `distr` (distributions), and the various quantum operations need expressions of types (α, α) `bounded`, α `vector`, and (α, β) `measurement`. Various predefined constants for constructing expressions of those types are described in the table below.

Predicates. Predicates (the post- and preconditions in qRHL judgments) are also interpreted by Isabelle/HOL. They have to be expressions of the type `mem2 subspace` (abbreviated `predicate`), with free classical program variables (indexed with `1` or `2`, i.e., if the program variable is `x`, then the expression may contain `x1` and `x2`). Here `mem2` is the type of pairs of memories, and thus `mem2 subspace` is the type with universe $\ell^2[V_1 V_2]$ if V_1, V_2 represent the indexed program variables.

Predicates can additionally contain quantum variables as arguments to specific constructions, e.g., `pauliX»`$[\![\mathbf{q}]\!]$ would refer to an Pauli-X operator on quantum variable $\mathbf{q}$.

Predicates can be constructed using the constants described in the tables below.

Types. The theory `QRHL` defines the following types. Some of those types are defined in Isabelle/HOL using `typedef`, others are only axiomatized, see `QRHL.thy` and the theories imported therein.

Type	**Meaning**

`bit`	The type of bits. This type is isomorphic to `bool`, but using the type `bit` can lead to more familiar notation in some cases because the constants 0 and 1 can be used. On bits, the operations $+$, $*$, $-$, / are defined modulo 2 (that is, `bit` is the finite field of size 2). In particular, the negation of x is written $x+1$ (not $-x$ which is equal to x). An implicit coercion is declared so that `bit` can be used where **nat** or `int` are expected.
α `distr`	The set of distributions over α. Recall from the preliminaries that in our context, distributions are functions $\mu : \alpha \to \mathbb{R}_{\geq 0}$ with $\sum_x \mu(x) \leq 1$. Expressions of this type occur on the right hand side of sample statements (e.g., `e` in `x <$ e;`).
α `vector`	Vectors in $\ell^2(\alpha)$. The type is endowed with the type class `normed_real_vector`, so operations such as $+$ or `norm` work as expected. Expressions of this type occur on the rhs of quantum initialization statements (e.g., `e` in `q <q e;`).
α `subspace`	Closed subspaces of $\ell^2(\alpha)$. This type is used mostly for constructing quantum predicates. It is endowed with the type class `complete_lattice`, thus it has operations such as $\sqcap$ (`inf`) for the intersection of two spaces, $\sqcup$ (`sup`) or $+$ for the sum of two spaces, `INF x:Z. f x` for the intersection of all spaces $f(x)$ for $x \in Z$, and $\leq$ for inclusion of subspaces. And `top` is the whole space $\ell^2(\alpha)$, and `0` and `bot` both refer to the zero-space 0.
`mem2`	The quantum part of pairs of memories. That is, if V_1, V_2 denote the set of all variables with indices 1 and 2, respectively, `mem2` represents $\mathsf{Type}^{\mathsf{set}}_{V_1^{\mathsf{qu}} V_2^{\mathsf{qu}}}$. This type is mainly used for defining the type `predicate`.
`predicate`	An abbreviation for `mem2 subspace`, that is, subspaces of $\ell^2(\mathsf{Type}^{\mathsf{set}}_{V_1^{\mathsf{qu}} V_2^{\mathsf{qu}}}) = \ell^2[V_1^{\mathsf{qu}} V_2^{\mathsf{qu}}]$. This is the type of quantum predicates. Expressions of this type occur in the pre- and postcondition of qRHL judgments, as well as in many subgoals generated by tactics.
(α, β) `bounded`	Bounded operators $\mathbf{B}(\alpha, \beta)$. Expressions of this type occur in quantum operation statements, e.g., `U` in "`on q apply U`". In that case, `U` should always describe an isometry. (See the description of quantum operation statements in Section 3.) Expressions of this type also occur in predicates, e.g., as an argument to `quantum_equality_full` or due to application of the `wp` tactic (implementing rule QApply1).
(α, β) `measurement`	Measurements $\mathbf{Meas}(\alpha, \beta)$. Expressions of this type occur in measurements statements, e.g., `M` in "`x <- measure q with M`".

α `variable`	Represents a program variable $\mathbf{q}$ with $\mathsf{Type}_{\mathbf{q}} = \alpha$. One can think of a variable of type α `variable` as a variable name, associated with type α. There are no constants for creating values of type α `variable`. Instead, by declaring a quantum variable using `quantum var q : T;` in the tool, `q1` and `q2` will automatically be declared to have type `T variable`.[6] Quantum variables are needed to specify registers when constructing predicates. (See, e.g., the description of the `lift` constant below.)
α `variables`	Tuples of program variables. When $\mathtt{q}_1, \ldots, \mathtt{q}_n$ are variables of types α_1 `variable`,...,α_n `variable`, then their tuple (constructed with the syntax $[\![\mathtt{q}_1, \ldots, \mathtt{q}_n]\!]$) has type $(\alpha_1 \times \cdots \times \alpha_n)$ `variables`. Having such a type is necessary for specifying certain constants that operate on list of quantum variables (e.g., `lift`) in a type-safe way.
`program`	A program. When a program `P` is declared with `program P := ...;` or `adversary P ...;`, then `P` will have type `program` in Isabelle/HOL expressions. `P` can then be used as an argument to the `Pr[...]` constant (see the table below). There are no other uses of this type in our development.
`program_state`	A program state. That is, an element of $\mathbf{T}^+_{cq}[V_1V_2]$ of trace 1, where V_1, V_2 denote the set of all variables with indices 1 and 2, respectively. This type is not interpreted in any way, there is are no constants for constructing program states. The only use is as an argument to the `Pr[...]` constant (see the table below), to refer to an unspecified but fixed quantum state (typically declared by `ambient var rho : program_state;`).

Constants. The theory `QRHL` defines the following constants for use in expressions and predicates. In many cases, there are several possible syntaxes for entering the same constant. We list all of them, the first being the one Isabelle/HOL will use for printing the constant. In many cases, the syntax contains special characters. These can be entered with the TeX input method of Emacs (which is automatically active in our ProofGeneral customization). In those cases we additionally mention the character sequences to be entered in ProofGeneral for getting the special characters (marked "How to input:" in the table below).

Name / syntax / type	**Meaning**
	Distributions
`supp` μ :: α `set` (for μ :: α `distr`)	The support $\operatorname{supp}\mu$ of the distribution μ.
`weight` μ :: *real* (for μ :: α `distr`)	The weight of the distribution, that is $\sum_x \mu(x)$. In particular, μ is total iff `weight` $\mu = 1$.
`map_distr` f μ :: β `distr` (for f :: $\alpha \Rightarrow \beta$ and μ :: α `distr`)	The distribution of $f(x)$ when x is μ-distributed. That is, $\nu(x) = \sum_{y \in f^{-1}(\{x\})} \mu(y)$ for $\nu :=$ `map_distr` f μ. In particular, the first and second marginal of a distribution μ on pairs are given by `map_distr fst` μ and `map_distr snd` μ, respectively.
`prob` μ x :: `real` (for μ :: α `distr` and x :: α)	The probability $\mu(x)$ of x according to distribution μ.

[6] `classical var x : T;` also declares values of type `T variable` in Isabelle, but those are not needed on the user level, they are used internally.

`uniform` M :: α `distr` (for M :: α `set`)	The uniform distribution on the set M if M is finite and non-empty. If M is infinite or empty, then `uniform` $M := 0$.
`Pr`$[v : P(\rho)]$ :: `real` (for v :: `bool`, `bit` and P :: `program` and ρ :: `program_state`)	The probability $\Pr[v : P(\rho)]$ that $v = 1$ or $v =$ `true` after execution of the program P with initial state ρ. Here v must be the name of a classical variable declared with `classical var` v `: bit;` (or `bool`). (An expression that merely evaluates to v is not allowed. For example: `Pr`$[$`fst` $(v, v) : P(\rho)]$ is not a synonym for `Pr`$[v : P(\rho)]$. This is because a special syntax translation operates on probability expressions.) P can be the name of a program declared using `program P := ...;` or `adversary P var ...;`. (But in the case of P, expressions that evaluate to a program are also admissible.) The constant `probability` is internally used for representing `Pr`$[v : P(\rho)]$. It should not be used directly.

Operators

A^* `adjoint` A :: (β, α) `bounded` (for A :: (α, β) `bounded`)	The adjoint A^* of A.
$A \cdot B$ `timesOp` A B :: (α, γ) `bounded` (for A :: (β, γ) `bounded` and B :: (α, β) `bounded`)	The product AB of operators A and B. The syntax $A \cdot B$ is overloaded. If Isabelle/HOL has trouble recognizing which meaning of $\cdot$ is intended, use `timesOp`, or provide additional type information for A and B. **How to input:** `\cdot`
$A \cdot \psi$ `applyOp` A ψ :: β `vector` (for A :: (α, β) `bounded` and ψ :: α `vector`)	The result $A\psi$ of applying the operator A to the vector ψ. The syntax $A \cdot \psi$ is overloaded. If Isabelle/HOL has trouble recognizing which meaning of $\cdot$ is intended, use `applyOp`, or provide additional type information for the A and ψ. **How to input:** `\cdot`
$A \cdot S$ `applyOpSpace` A S :: β `subspace` (for A :: (α, β) `bounded` and S :: α `subspace`)	The result $AS = \{A\psi : \psi \in S\}$ of applying the operator A to the subspace ψ. The syntax $A \cdot S$ is overloaded. If Isabelle/HOL has trouble recognizing which meaning of $\cdot$ is intended, use `applyOpSpace`, or provide additional type information for the A and S. **How to input:** `\cdot`
`idOp` :: (α, α) `bounded`	The identity operator id on $\ell^2(\alpha)$.
`addState` ψ :: $(\beta, \beta \times \alpha)$ `bounded` (for ψ :: α `vector`)	The operator mapping ϕ to $\phi \otimes \psi$. (Where $\otimes$ denotes a positional tensor product, not the labeled tensor product defined in Section 2 in [9].)
`unitary` A :: `bool` (for A :: (α, β) `bounded`)	True iff A is unitary.
`isometry` A :: `bool` (for A :: (α, β) `bounded`)	True iff A is an isometry.

`isProjector` A :: `bool` (for A :: (α,α) `bounded`)	True iff A is a projector.
`Proj` S :: (α,α) `bounded` (for S :: α `subspace`)	The projector onto subspace S.
`hadamard`,`pauliX`,`pauliY`,`pauliZ` :: (`bit`, `bit`) `bounded`	Hadamard, or Pauli X, Y, Z operators, respectively.
`CNOT` :: (`bit`×`bit`, `bit`×`bit`) `bounded`	Controlled-not on two qubits (first qubit is the control)
$A \otimes B$ `tensor` A B `tensorOp` A B :: (α,β) `bounded` (for A :: α `bounded` and B :: β `bounded`)	The (positional) tensor product $A \otimes B$ of operators. (Not the labeled one between $\mathbf{B}(V)$ and $\mathbf{B}[W]$ described in the preliminaries of [9]. That is, $A \otimes B \neq B \otimes A$.) The notations $A \otimes B$ and `tensor` A B are overloaded. If Isabelle/HOL has trouble recognizing which meaning is intended, use `tensorOp`, or provide additional type information for A or B. **How to input:** `\otimes`
`comm_op` :: $(\alpha \times \beta, \beta \times \alpha)$ `bounded`	The canonical isomorphism between $\ell^2(X \times Y)$ and $\ell^2(Y \times X)$. That is, the operator mapping $\|x,y\rangle$ to $\|y,x\rangle$.
`assoc_op` :: $(\alpha\times(\beta\times\gamma),(\alpha\times\beta)\times\gamma)$`bounded`	The canonical isomorphism between $\ell^2(X\times(Y\times Z))$ and $\ell^2((X\times Y)\times Z)$. That is, the operator mapping $\|x,(y,z)\rangle$ to $\|(x,y),z\rangle$. Note that in Isabelle/HOL, $\alpha \times (\beta \times \gamma)$ is the same type as $\alpha \times \beta \times \gamma$ but not the same as $(\alpha \times \beta) \times \gamma$. If we identify all those types, then `assoc_op` is the identity operator.

States

$\|x\rangle$ `ket` x :: α `vector` (for x :: α)	The basis state $\|x\rangle$ of $\ell^2(\alpha)$. **How to input:** `\rangle`
`EPR` :: (`bit` × `bit`) `vector`	The state $\frac{1}{\sqrt{2}}\|00\rangle + \frac{1}{\sqrt{2}}\|11\rangle$.
$\psi \otimes \phi$ `tensor` ψ ϕ `tensorVec` ψ ϕ :: (α,β) `vector` (for ψ :: α `vector` and ϕ :: β `vector`)	The (positional) tensor product $\psi \otimes \phi$ of vectors. (Not the labeled one between $\ell^2[V]$ and $\ell^2[W]$ described in the preliminaries of [9]. That is, $\psi \otimes \phi \neq \phi \otimes \psi$.) The notations $\psi \otimes \phi$ and `tensor` ψ ϕ are overloaded. If Isabelle/HOL has trouble recognizing which meaning is intended, use `tensorVec`, or provide additional type information for ψ or ϕ. **How to input:** `\otimes`

Quantum variables

$[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$ $[\|\mathbf{q}_1, \ldots, \mathbf{q}_n\|]$ :: $(\alpha_1 \times \cdots \times \alpha_n)$ `variables` (for $\mathbf{q}_i :: \alpha_i$ `variable`)	A typed tuple of quantum variables. Constants that can be applied to several quantum variables expect a typed tuple of quantum variables because their result type depends on the types of all involved quantum variables. For example the Isabelle/HOL expression $[\![\mathbf{q}_1, \mathbf{q}_2]\!] \equiv_{\mathsf{quant}} [\![\mathbf{q}'_1]\!]$ expresses the quantum equality $\mathbf{q}_1\mathbf{q}_2 \equiv_{\mathsf{quant}} \mathbf{q}'_1$ and it is well-typed iff $\mathsf{Type}_{\mathbf{q}_1} \times \mathsf{Type}_{\mathbf{q}_2} = \mathsf{Type}_{\mathbf{q}'_1}$. Typed quantum variables allow Isabelle/HOL to check those type conditions. **How to input:** `\llbracket, \rrbracket`
A»Q $A >> Q$ `lift` A Q `liftOp` A Q :: (`mem2`, `mem2`) `bounded` (for $A ::$ (α, α) `bounded` and $Q ::$ α `variables`)	The operator A»$Q := U_{vars,Q} A U^*_{vars,Q} \otimes id_{V_1^{\mathsf{qu}} V_2^{\mathsf{qu}} \setminus Q}$. Intuitively, » takes an operator A on $\ell^2(\alpha)$, and returns the operator A»Q on $\ell^2[V_1V_2]$ that corresponds to applying A on the quantum variables $Q \subseteq V_1V_2$. The syntax A»Q and `lift` is overloaded. If Isabelle/HOL has trouble recognizing which meaning of » or `lift` is intended, use `liftOp`, or provide additional type information for the lhs A. **How to input:** `\frqq`
S»Q $S >> Q$ `lift` S Q `liftSpace` S Q :: `predicate` (for $A ::$ α `subspace` and $Q ::$ α `variables`)	The subspace S»$Q := U_{vars,Q} S \otimes \ell^2[V_1V_2 \setminus Q]$. Intuitively, » takes a subspace S of $\ell^2(\alpha)$, and returns the subspace S»Q of $\ell^2[V_1V_2]$ that corresponds to the state of variables Q being in subspace S. The syntax S»Q and `lift` is overloaded. If Isabelle/HOL has trouble recognizing which meaning of » or `lift` is intended, use `liftSpace`, or provide additional type information for the lhs S. **How to input:** `\frqq`
`distinct_qvars` Q :: `bool` (for $Q ::$ α `variables`)	True if the variables in the quantum variable tuple Q are all distinct. To automatically simplify statements of this form in an accompanying Isabelle theory, it is recommended to add a fact of the form `declared_qvars` $[\![\ldots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.
`colocal` P Q `colocal_pred_qvars` P Q :: `bool` (for $P ::$ `predicate` and $Q ::$ α `variables`)	True iff the predicate P is X-local for some set of variables with $X \cap Q = \varnothing$, and no variable occurs twice in Q. The syntax `colocal` P Q is overloaded. If Isabelle/HOL has trouble recognizing which meaning of `colocal` is intended, use `colocal_pred_qvars`, or provide additional type information for P and Q. To automatically simplify statements of this form in an accompanying Isabelle theory, it is recommended to add a fact of the form `declared_qvars` $[\![\ldots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.

`colocal` A Q `colocal_op_qvars` A Q :: `bool` (for A :: (`mem2`, `mem2`) `bounded` and Q :: α `variables`)	True iff the operator A is X-local for some set of variables with $X \cap Q = \varnothing$, and no variable occurs twice in Q. The syntax `colocal` A Q is overloaded. If Isabelle/HOL has trouble recognizing which meaning of `colocal` is intended, use `colocal_op_qvars`, or provide additional type information for A and Q. To automatically simplify statements of this form in an accompanying Isabelle theory, it is recommended to add a fact of the form `declared_qvars` $[\![\ldots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.
`colocal` A P `colocal_op_pred` A P :: `bool` (for A :: (`mem2`, `mem2`) `bounded` and P :: `predicate`)	True if the operator A is X-local and the predicate P is Y-local for some sets X, Y of quantum variables with $X \cap Y = \varnothing$. The syntax `colocal` A P is overloaded. If Isabelle/HOL has trouble recognizing which meaning of `colocal` is intended, use `colocal_op_pred`, or provide additional type information for A and P. To automatically simplify statements of this form in an accompanying Isabelle theory, it is recommended to add a fact of the form `declared_qvars` $[\![\ldots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.
`declared_qvars` $[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$ `declared_qvars` $[\vert\mathbf{q}_1, \ldots, \mathbf{q}_n\vert]$ :: `bool` (for $\mathbf{q}_i$ :: α_i `variable`)	Informally, indicates that all $\mathbf{q}_i$ are quantum variables declared in the tool. All $\mathbf{q}_i$ must be free Isabelle variables referring directly to quantum variables (i.e., not bound variables, nor is it permitted to, e.g., define x as an alias for $\mathbf{q}$ and then use x here). Formally, this is an abbreviation for `variable_name` $\mathbf{q}_1$ = $s_1 \wedge \cdots \wedge$ `variable_name` $\mathbf{q}_n$ = s_n, where s_i is a string literal containing the name of the variable $\mathbf{q}_i$. The simplifier can use these statements to automatically prove `distinct_qvars` $[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$ and various statements of the form `colocal`.... When reasoning in Isabelle directly (in an accompanying theory), it is advisable to add the assumption `declared_qvars` $[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$ (where $\mathbf{q}_i$ are quantum variables declared using `quantum var ...` in our tool) as an assumption to lemmas that are proven in Isabelle, and to add this assumption to the Isabelle simplifier. See `Teleport_Terse.thy` and `Teleport.thy` for examples. When invoking the simplifier from the tool via the `simp` tactic, it is not necessary to add those assumptions because the `simp` tactic already adds it automatically. In particular, ambient subgoals of the form `declared_qvars` $[\![\ldots]\!]$ are solved automatically by the `simp` tactic.

Subspaces & predicates

`span` M :: α `subspace` (for M :: α `vector set`)	The span "span M" of the states in M.

$\mathfrak{Cla}[b]$ `Cla`$[b]$ `classical_subspace` b :: `predicate` (for b :: `bool`)	The predicate $\mathfrak{Cla}[b] \subseteq \ell^2[V_1 V_2]$. This allows to encode predicates about classical variables within quantum predicates. **How to input:** There is no input method support for entering $\mathfrak{Cla}$. Use the `Cla`$[\dots]$ syntax instead.
`quantum_equality_full` $A_1\, Q_1\, A_2\, Q_2$ :: `predicate` (for A_1 :: (α,γ) `bounded` and Q_1 :: α `variables` and A_2 :: (β,γ) `bounded` and Q_2 :: β `variables`)	The quantum equality predicate $A_1 Q_1 \equiv_{\mathsf{quant}} A_2 Q_2$. (Definition 26 in [9])
$Q_1 \equiv\mathfrak{q}\ Q_2$ Q_1 `==q` Q_2 `quantum_equality` $Q_1\ Q_2$ `Qeq`$[\mathbf{q}_1,\dots,\mathbf{q}_n = \mathbf{q}'_1,\dots,\mathbf{q}'_m]$ :: `predicate` (for Q_1 :: α `variables` and Q_2 :: α `variables`)	Quantum equality $Q_1 \equiv_{\mathsf{quant}} Q_2$. (Definition 27 in [9]) This is an abbreviation for `quantum_equality_full idOp` Q_1 `idOp` Q_2. (That is, Isabelle/HOL internally expands this abbreviation whenever it encounters it.) The syntax `Qeq`$[\mathbf{q}_1,\dots,\mathbf{q}_n = \mathbf{q}'_1,\dots,\mathbf{q}'_m]$ is a convenience input syntax for inputting $[\![\mathbf{q}_1,\dots,\mathbf{q}_n]\!] \equiv\mathfrak{q}\ [\![\mathbf{q}'_1,\dots,\mathbf{q}'_m]\!]$. The variables $\mathbf{q}_i, \mathbf{q}'_i$ must have types α_i, α'_i such that $\alpha_1 \times \cdots \times \alpha_n = \alpha'_1 \times \cdots \times \alpha'_m$. **How to input:** There is no input method support for `q`. Use, e.g., the `==q` syntax instead of $\equiv\mathfrak{q}$.
$P \div \psi \gg Q$ `space_div` $P\ \psi\ Q$ :: `predicate` (for P :: `predicate` and ψ :: α `vector` and Q :: α `variables`)	The quantum predicate $(P \div U_{vars,Q}\psi) \otimes \ell^2[Q]$. Note that the only place where $\div$ appear in our qRHL rules is in rule QInit1, where it appears in an expression of the form $(P \div U_{vars,Q}\psi) \otimes \ell^2[Q]$. Because of this it is more convenient in the tool to directly define this combination as a single constant instead of breaking it down into several (more difficult to type) building blocks. **How to input:** `\div, \frqq`
`ortho` S :: α `subspace` (for S :: α `subspace`)	Orthogonal complement $S^\perp$ of S.
$S \otimes T$ `tensor` $S\ T$ `tensorSpace` $S\ T$ :: (α,β) `subspace` (for S :: α `subspace` and T :: β `subspace`)	The (positional) tensor product $S \otimes T$ of subspaces. (Not the labeled one between $\ell^2[V]$ and $\ell^2[W]$ described in the preliminaries of [9]. That is, $S \otimes T \neq T \otimes S$.) The notations $S \otimes T$ and `tensor` $S\ T$ are overloaded. If Isabelle/HOL has trouble recognizing which meaning is intended, use `tensorSpace`, or provide additional type information for S or T. **How to input:** `\otimes`

Measurements

`computational_basis` :: (α,α) `measurement`	A projective measurement on $\ell^2(\alpha)$ in the computational basis.
`mtotal` M :: `bool` (for M :: (α,β) `measurement`)	True iff the measurement M is total.

`mproj` M x :: (β, β) `bounded` (for $M :: (\alpha, \beta)$ `measurement` and $x :: \alpha$)	The projector $M(x)$ corresponding to outcome x of the projective measurement M.

5 Tactics

In this section, we document all tactics supported by our tool. The tactics are not in one-to-one correspondence with the rules from Section 5 (for example, many tactics implement a combination of some rule with the Seq or Conseq rule). Yet, most rules can be recovered as special cases of the tactics. (E.g., the rule Sample1 can be implemented as the tactic sequence `wp left. skip. simp.`) Some rules are not implemented in their full generality (e.g., Frame is implemented by `equal` which does not take into account readonly variables). Rules that are not yet implemented in the tool are: Sym, QrhlElim (but we have QrhlElimEq), JointSample, JointIf, While1, JointWhile, JointMeasure, JointMeasureSimple.

In the description of the rules, we use Isabelle/HOL syntax for expressions (in particular, for pre- and postconditions) because that is the syntax used in our tool. The reader should keep this in mind when comparing the rules described in this section with those from Section 5 in [9]. See Section 6 for a description of the constants used in Isabelle/HOL syntax.

Whenever we state a rule describing the operation of a tactic, the preconditions of the rule are the subgoals created by the tactic. Any other preconditions the rule may have (i.e., conditions that the tactic checks immediately instead of creating a subgoal) are mentioned in the text accompanying the rule.

Tactic admit

Solves the current subgoal without checking. This tactic is *not sound*, it can be used to prove any theorem. It is intended for experimentation and proof development (to get a subgoal out of the way temporarily and focus on other subgoals first).

Tactic byqrhl

Transforms a goal of the form $\mathtt{Pr}[\mathbf{x} : P(\rho)] = \mathtt{Pr}[\mathbf{x}' : P'(\rho)]$ into a qRHL subgoal. (Also works for $\leq$ or $\geq$ instead of $=$.)

Here $\mathbf{x}, \mathbf{x}'$ must be classical variables of type `bit` or `bool`, and P, P' must be the names of programs that have been declared using the `program` or the `adversary` command.

The tactic implements the following rule:

$$\frac{\{\mathfrak{Cla}[\mathbf{y}_1^{(1)} = \mathbf{y}_2^{(1)} \wedge \cdots \wedge \mathbf{y}_1^{(n)} = \mathbf{y}_2^{(n)}] \sqcap [\![\mathbf{q}_1^{(1)}, \ldots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \ldots, \mathbf{q}_2^{(m)}]\!]\}\mathtt{call}\ P \sim \mathtt{call}\ P'\{\mathfrak{Cla}[\mathbf{x}_1 \leftrightarrow \mathbf{x}_2']\}}{\mathtt{Pr}[\mathbf{x} : P(\rho)] = \mathtt{Pr}[\mathbf{x}' : P'(\rho)]}$$

Here $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)}$ are the free classical variables of P, P'. And $\mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(m)}$ are the free quantum variables of P, P'. (Including those of any programs included recursively within P, P' via `call`.)

If $\mathbf{x}, \mathbf{x}'$ are of type `bit`, the precondition contains $(\mathbf{x}_1 = 1) \leftrightarrow (\mathbf{x}_2' = 1)$ instead of $\mathbf{x}_1 \leftrightarrow \mathbf{x}_2$. If the conclusion contains $\leq$ or $\geq$ instead of $=$, then $\leftrightarrow$ is replaced by $\rightarrow$ or $\leftarrow$, respectively. If $m = 0$, then $[\![\mathbf{q}_1^{(1)}, \ldots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_n^{(1)}, \ldots, \mathbf{q}_n^{(m)}]\!]$ is replaced by `top`.

The rule is a special case of rule QrhlElimEq.

Tactic equal

Converts a subgoal of the form $\{A\}\mathbf{c}; s \sim \mathbf{c}'; s\{B\}$ where s is a single statement into a subgoal $\{A\}\mathbf{c} \sim \mathbf{c}'\{C\}$ with suitably updated postcondition C. (Note: an if- or while-statement counts as a single statement.)

The postcondition C is derived as follows:

Let $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ be the free classical variables of s, and $\mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(m)}$ be the free quantum variables of s. (Including those of any programs included recursively within s via `call` statements.)

First, we rewrite the postcondition B is several steps. This is in order to get a new postcondition C that does not share any free variables with s (if possible) while strengthening the postcondition as little as possible.

B' is B with occurrences of $\mathbf{x}_1^{(i)} = \mathbf{x}_2^{(i)}$ for all i, and occurrences of $[\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$ removed. (In such a way that $B' \sqcap \mathfrak{Cla}[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \wedge \dots \wedge \mathbf{x}_1^{(n)} = \mathbf{x}_2^{(n)}] \sqcap [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$ is a subset of B.)

Let $i_1, \dots, i_k$ be all indices such that both $\mathbf{x}_1^{(i)}$ and $\mathbf{x}_2^{(i)}$ are free variables in B'. Let

$$B'' := \mathfrak{Cla}\big[\neg(\mathbf{x}_1^{(i_1)} = \mathbf{x}_2^{(i_1)} \wedge \dots \wedge \mathbf{x}_1^{(i_k)} = \mathbf{x}_2^{(i_k)})\big] + B'.$$

Note that also $B'' \sqcap \mathfrak{Cla}\big[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \wedge \dots \wedge \mathbf{x}_1^{(n)} = \mathbf{x}_2^{(n)}\big] \sqcap [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$ is a subset of B.

Let

$$B''' := \texttt{INF}\ X_1 X_2.\ B''$$

where $X_1 X_2$ are all classical variables $\mathbf{x}_j^{(i)}$ (for $j = 1, 2$) such that $\mathbf{x}_j^{(i)}$ is a free variable of s, and $\mathbf{x}_j^{(i)}$ is free in B''.

Let

$$C := B''' \sqcap \mathfrak{Cla}[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \wedge \dots \wedge \mathbf{x}_1^{(n)} = \mathbf{x}_2^{(n)}] \sqcap [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!].$$

Note that C is a subset of B, and note that $fv(B''')^{\mathsf{cl}} \cap fv(s) = \varnothing$.

With this definitions, the rule implemented by the tactic is:

$$\frac{\texttt{colocal}\ B'\ [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}, \mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!] \qquad \{A\}\mathbf{c} \sim \mathbf{c}'\{C\}}{\{A\}\mathbf{c}; s \sim \mathbf{c}'; s\{B\}}$$

This rule is shown as follows: From rule Equal, we have $\{Eqs\}s \sim s\{Eqs\}$ where $Eqs := \mathfrak{Cla}[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \wedge \dots \wedge \mathbf{x}_1^{(n)} = \mathbf{x}_2^{(n)}] \sqcap [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$. With rule Frame, we get $\{B''' \sqcap Eqs\}s \sim s\{B''' \sqcap Eqs\}$. (We have that s does not share variables with B''' as required by rule Frame since $fv(B''')^{\mathsf{cl}} \cap fv(P) = \varnothing$ and since $\texttt{colocal}\ B'\ [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}, \mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$ implies $\texttt{colocal}\ B'''\ [\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}, \mathbf{q}_2^{(1)}, \dots, \mathbf{q}_2^{(m)}]\!]$.) With rule Seq, and using $C = B''' \sqcap Eqs$, we get $\{A\}\mathbf{c}; \texttt{call}\ P \sim \mathbf{c}'; \texttt{call}\ P\{B''' \sqcap Eqs\}$. And with rule Conseq, we get $\{A\}\mathbf{c}; \texttt{call}\ P \sim \mathbf{c}'; \texttt{call}\ P\{B\}$.

The first subgoal intuitively means that the quantum variables from the program do not occur in the postcondition (except within $[\![\mathbf{q}_1^{(1)}, \dots, \mathbf{q}_1^{(m)}]\!] \equiv_{\mathfrak{q}} [\![\mathbf{q}_n^{(1)}, \dots, \mathbf{q}_n^{(m)}]\!]$). This subgoal can usually be discharged with "`simp.`".

We explain shortly the motivation behind the construction of the new postcondition C. We need that C does not contain any of the program variables of s (except within Eqs). (Otherwise we cannot apply the Frame rule.) And furthermore, we need that C is a subset of B. And finally, C must be of the form $B''' \sqcap \texttt{Eqs}$ for some B'''.

Our definition of C is an attempt of finding a reasonably large B''' (and thus C) that satisfies those constraints.

For this, we start with B, and remove as many terms containing program variables of s from B as we can. Namely, since $C = B''' \sqcap Eqs$, we can remove all equations from B that occur in Eqs anyway without weakening the predicate. This leads to B'. If there remain any quantum variables of s in B', there seems to be nothing we can do. However, if there are classical variables $X_1 X_2$ of s remaining, we can get rid of them by replacing B' by $\texttt{INF}\ X_1 X_2.\ B'$. (If B' is classical, this is equivalent to all-quantifying over $X_1 X_2$.) This is the idea behind the definition of B'''. However, this would mean that inside $\texttt{INF}\ X_1 X_2.\ B'$, the information is lost that $\mathbf{x}_1^{(i)} = \mathbf{x}_2^{(i)}$. (E.g., if $B = \mathfrak{Cla}[\mathbf{x}_1 \geq \mathbf{x}_2]$, then $\texttt{INF}\ \mathbf{x}_1 \mathbf{x}_2.\ B'$ is equivalent to $\mathfrak{Cla}[\forall \mathbf{x}_1 \mathbf{x}_2.\ \mathbf{x}_1 \geq \mathbf{x}_2]$ which is false.) To recover this information, we instead define B''' as

$$\texttt{INF}\ \mathbf{x}_1 \mathbf{x}_2.\ \mathfrak{Cla}\big[\neg(\mathbf{x}_1^{(i_1)} = \mathbf{x}_2^{(i_1)} \wedge \dots \wedge \mathbf{x}_1^{(i_k)} = \mathbf{x}_2^{(i_k)})\big] + B'.$$

For classical $B' = \mathfrak{Cla}[e]$, this is equivalent to

$$\mathfrak{Cla}\Big[\forall X_1 X_2.\ (\mathbf{x}_1^{(i_1)} = \mathbf{x}_2^{(i_1)} \wedge \dots \wedge \mathbf{x}_1^{(i_k)} = \mathbf{x}_2^{(i_k)}) \longrightarrow B'\Big].$$

Then B''' can be satisfiable even if B relies on the equality of the classical variables. (For example, if $B = \mathfrak{Cla}[\mathbf{x}_1 \geq \mathbf{x}_2]$, we have $B''' = \big(\texttt{INF}\ \mathbf{x}_1 \mathbf{x}_2.\ \mathfrak{Cla}[\neg \mathbf{x}_1 = \mathbf{x}2] + \mathfrak{Cla}[\mathbf{x}_1 \geq \mathbf{x}_2]\big) = \mathfrak{Cla}[\forall \mathbf{x}_1 \mathbf{x}_2.\ \mathbf{x}_1 = \mathbf{x}_2 \longrightarrow \mathbf{x}_1 \geq \mathbf{x}_2]$ which is true.) We only add those equalities in B'' that actually may be relevant for B', to keep the term small.

Tactic case

When invoked as "`case` $z := e$.", it replaces the subgoal $\{A\}\mathbf{c} \sim \mathbf{d}\{B\}$ by $\{\mathfrak{Cla}[z = e] \sqcap A\}\mathbf{c} \sim \mathbf{d}\{B\}$. The variable z must be a declared as an ambient variable that is not contained in $\mathbf{c}, \mathbf{d}, e$ or in the code of any program declared with the `program` command.

$$\frac{\{\mathfrak{Cla}[z = e] \sqcap A\}\mathbf{c} \sim \mathbf{d}\{B\}}{\{A\}\mathbf{c} \sim \mathbf{d}\{B\}}$$

The tactic is justified by rule Case. Note that rule Case would add an additional all-quantifier $\forall z$ to the subgoal. However, since all ambient variables are implicitly all-quantified, the all-quantifier can be omitted.

Tactic clear

When invoked as "`clear` n" for some integer $n \geq 1$, it removes the n-th assumption from the current subgoal. For qRHL subgoals, assumptions are explicitly listed and numbered in the tool. For ambient subgoals of the form $A_1 \to \cdots \to A_m \to B$, A_n is considered to be the n-th assumption.

$$\frac{A_1 \to \ldots A_{n-1} \to A_{n+1} \to \cdots \to A_m \to B}{A_1 \to \cdots \to A_m \to B}$$

Tactic casesplit

When invoked as "`casesplit` e." with a Boolean expression e, the current subgoal G is replaced by two subgoals $e \to G$ and $\neg e \to G$. This works for qRHL subgoals and ambient logic subgoals.

$$\frac{e \to G \qquad \neg e \to G}{G}$$

Tactic conseq

When invoked as "`conseq pre:` C.", it rewrites the precondition of the current qRHL subgoal to become C. When invoked as "`conseq post:` C.", it rewrites the postcondition of the current qRHL subgoal to become C. C must be an Isabelle/HOL expression of type `predicate`.

That is, one of the following two rules is applied (left for `pre`, right for `post`):

$$\frac{A \leq C \qquad \{C\}\mathbf{c} \sim \mathbf{d}\{B\}}{\{A\}\mathbf{c} \sim \mathbf{d}\{B\}} \qquad \frac{C \leq B \qquad \{A\}\mathbf{c} \sim \mathbf{d}\{C\}}{\{A\}\mathbf{c} \sim \mathbf{d}\{B\}}$$

Both rules are special cases of rule Conseq.

Tactic fix

When invoked as "`fix` z.", replaces a goal of the form $\forall x.\, e$ by $e\{z/x\}$, i.e., e with occurrences of x replaced by z. The variable z must be a declared as an ambient variable, and it must not occur free in e or in the code of any program declared with the `program` command.

$$\frac{e\{z/x\}}{\forall x.\, e}$$

This rule is justified by the fact that free ambient variables are implicitly all-quantified.

Tactic inline

When invoked as "`inline` P." it replaces all occurrences of `call` P`;` in the current subgoal by the code of P. Here P must be a program defined by `program` P `:= {...}.` The current goal must be a qRHL subgoal.

Logically, this does not change the subgoal since `call` P`;` is just an abbreviation for the code of P.

Tactic rnd

Converts a subgoal of the form $\{A\}\,\mathbf{c};\mathbf{x}\ \texttt{<\$}\ e\ \sim\ \mathbf{c}';\mathbf{x}'\ \texttt{<\$}\ e'\,\{B\}$ (i.e., ending in a sampling on both sides) into a subgoal $\{A\}\mathbf{c}\sim\mathbf{c}'\{C\}$ with suitably updated postcondition C.

Specifically, if invoked as "`rnd.`", the new postcondition will be $C := \mathfrak{Cla}[e_1 = e_2'] \sqcap (\texttt{INF}\ z \in \texttt{supp}\ e_1.\ B')$ where $e_1 := \mathrm{idx}_1\, e$ (all free classical variables in e indexed with 1), and $e_2' := \mathrm{idx}_2\, e$ (all free classical variables in e' indexed with 2), and $B' := B\{z/\mathbf{x}_1, z/\mathbf{x}_2'\}$ (i.e., all occurrences of $\mathbf{x}_1$ and $\mathbf{x}_2$ replaced by a fresh variable z).

Informally, C requires that e and e' are the same distribution, and B holds for any $\mathbf{x}_1 = \mathbf{x}_2'$ in the support of e. That is, the syntax "`rnd.`" is to be used in the common case when both programs end with the same sampling, and we want the two samplings to be "in sync", i.e., to return the same value.

The variables $\mathbf{x}$ and $\mathbf{x}'$ must have the same type in this case.

That is, "`rnd.`" implements the following rule:

$$\frac{\big\{A\big\}\mathbf{c}\sim\mathbf{c}'\big\{\mathfrak{Cla}[e_1 = e_2'] \sqcap \big(\texttt{INF}\ z \in \texttt{supp}\ e_1.\ B\{z/\mathbf{x}_1, z/\mathbf{x}_2'\}\big)\big\}}{\{A\}\,\mathbf{c};\mathbf{x}\ \texttt{<\$}\ e\ \sim\ \mathbf{c}';\mathbf{x}'\ \texttt{<\$}\ e'\,\{B\}} \qquad \text{where } e_1 := \mathrm{idx}_1\, e,\ e_2' := \mathrm{idx}_2\, e'$$

This rule is a consequence of rule JointSample and rule Seq: From rule JointSample (with $f :=$ `map_distr` $(\lambda z.\,(z,z))\ e_1$ and some simplifying), we get

$$\{\mathfrak{Cla}[e_1 = e_2'] \sqcap \big(\texttt{INF}\ z \in \texttt{supp}\ e_1.\ B\{z/\mathbf{x}_1, z/\mathbf{x}_2'\}\big)\}\mathbf{x}\ \texttt{<\$}\ e \sim \mathbf{x}'\ \texttt{<\$}\ e'\{B\}.$$

With rule Seq, the conclusion of the rule follows.

The second way of invoking the tactic is "`rnd` $\mathbf{x},\mathbf{x}'$ `<-` f`.`" Here $\mathbf{x},\mathbf{x}'$ must be the same variables as in the sampling statements in the subgoal.

In this case, the new subgoal will be $\{A\}\mathbf{c}\sim\mathbf{c}'\{C\}$ with

$$C := \mathfrak{Cla}[\texttt{map\_distr fst}\ f = e_1 \wedge \texttt{map\_distr snd}\ f = e_2'] \sqcap \big(\texttt{INF}\ (\mathbf{x}_1,\mathbf{x}_2') \in \texttt{supp}\ f.\ B\big)$$

where $e_1 := \mathrm{idx}_1\, e$ (all variables in e indexed with 1), and $e_2' := \mathrm{idx}_2\, e'$ (all variables in e' indexed with 2).

Informally, C says f has marginals e and e', and the postcondition B holds for any possible $\mathbf{x}_1,\mathbf{x}_2'$ in the support of f. This variant is used if the variables $\mathbf{x},\mathbf{x}'$ in the two programs are sampled according to potentially different distributions, and we want to establish a specific relationship between those variables after sampling (the relationship is encoded in the choice of f).

That is, the tactic "`rnd` $\mathbf{x},\mathbf{x}'$ `<-` f`.`" implements the following rule:

$$\frac{\big\{A\big\}\mathbf{c}\sim\mathbf{c}'\big\{\mathfrak{Cla}[\texttt{map\_distr fst}\ f = e_1 \wedge \texttt{map\_distr snd}\ f = e_2'] \sqcap \big(\texttt{INF}\ (\mathbf{x}_1,\mathbf{x}_2') \in \texttt{supp}\ f.\ B\big)\big\}}{\{A\}\,\mathbf{c};\mathbf{x}\ \texttt{<\$}\ e\ \sim\ \mathbf{c}';\mathbf{x}'\ \texttt{<\$}\ e'\,\{B\}}$$

$$\text{where } e_1 := \mathrm{idx}_1\, e,\ e_2' := \mathrm{idx}_2\, e'$$

The rule is an immediate consequence of rule JointSample and rule Seq.

Readers familiar with EasyCrypt may notice that their `rnd`-tactic takes very different arguments. Namely, in EasyCrypt, one can invoke the tactic as `rnd` F G where F and G are isomorphisms between the distributions e_1, e_2'. The EasyCrypt behavior can be recovered in our tool by invoking `rnd` $\mathbf{x},\mathbf{x}'$ `<- map_distr` $(\lambda z.\,(z, F\,z))\ e_1$. (Instead of the condition that F is an isomorphism between the distributions, our tactic will have the equation `map_distr snd map_distr` $(\lambda z.\,(z, F\,z))\ e_1 = e_2'$ in the resulting precondition, which follows from the fact that F is an isomorphism.) Our tactic is more general though, since we can also handle the case where the distributions are not isomorphic. For example, we can show the judgment $\{\texttt{top}\}\mathbf{x}\ \texttt{<\$}\ d;\sim\mathbf{x}\ \texttt{<\$}\ \texttt{map\_distr}\ (\lambda z.\ z*z)\ d;\{\mathfrak{Cla}[\mathbf{x}_1 * \mathbf{x}_1 = \mathbf{x}_2]\}$ (see the contributed file `rnd.qrhl`[7]) which does not seem easily possible in EasyCrypt.

Tactic rule

When invoked as "`rule` l" on an ambient subgoal, it applies the rule l to the current subgoal. That is, l is assumed to be the name of an Isabelle lemma of the form $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$, where B matches the current goal (i.e., $B\sigma$ is the current goal for some substitution σ). The current goal is then replaced by goals $A_1\sigma$, …, $A_n\sigma$.

[7] `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/rnd.qrhl`, and bundled with the tool.

This tactic is particularly useful for delegating subproofs to Isabelle/HOL. For example, if the current subgoal is an inequality of predicates that the `simp`-tactic cannot solve, then the subgoal can be copied to the accompanying Isabelle/HOL theory and proven there as a lemma l (possibly with some preconditions of the form `distinct_qvars` $[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$ that will then become new subgoals in the tool and can be resolved using the `simp`-tactic).

Tactic seq

When invoked as "`seq` i j`:` C", the tactic applies the rule

$$\frac{\{A\}s_1;\ldots;s_i \sim s'_1;\ldots;s_j\{C\} \qquad \{C\}s_{i+1};\ldots;s_n \sim s'_{j+1};\ldots;s_m\{B\}}{\{A\}s_1;\ldots;s_n \sim s'_1;\ldots;s'_m\{B\}}$$

That is, it splits off the first i statements on the left and the first j statements on the right of the current qRHL subgoal, and uses the argument C as the invariant to use in the middle.

If-statements count as single statements, even if their bodies contain multiple statements.

The rule is an immediate consequence of rule Seq.

Tactic simp

When invoked as "`simp` l_1 `...` l_n`.`", it runs the Isabelle/HOL simplifier on the current goal, resulting in one or zero subgoals.

More precisely, if the current goal is an ambient logic statement, the simplifier is applied directly. If the current goal is a qRHL judgment, the simplifier is applied to the precondition, the postcondition, and all assumptions (i.e., to all P_i if the current goal is $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow \{A\}\mathbf{c} \sim \mathbf{d}\{B\}$).

If the result is a trivial statement, the subgoal is removed. (Trivial statements are: `True`, qRHL judgments where one assumption is `False`, and qRHL judgments where the precondition is `bot`.)

The arguments $l_1, \ldots, l_n$ refer to names of Isabelle/HOL theorems. These are passed to the simplifier as additional simplification rules. They can either refer to theorems shown in Isabelle/HOL (e.g., in the theories included in Isabelle/HOL, in `QRHL.thy`, or in the accompanying theory loaded using the `isabelle TheoryName.` command), or to lemmas proven within the current proof script (when the goal was stated using `lemma` l_i`: ...`). These arguments are optional, the most common form of invoking the tactic is simply `simp.`

When invoked as "`simp !` l_1 `...` l_n`.`", the tactic behaves the same but fails unless the subgoal is solved and removed.

Tactic skip

Converts a qRHL subgoal $\{A\}\mathbf{skip} \sim \mathbf{skip}\{B\}$ into an ambient logic subgoal.

$$\frac{A \leq B}{\{A\}\mathbf{skip} \sim \mathbf{skip}\{B\}}$$

This rule is an immediate consequence of rules Skip and Conseq.

Tactic swap

This tactic swaps the last two statements on the left or the right side of a qRHL subgoal, assuming those statements are independent (have no joint variables). Specifically, when invoked as `swap left`, it applies the rule

$$\frac{\{A\}s_1;\ldots;s_{n-2};s_n;s_{n-1} \sim \mathbf{c}\{B\}}{\{A\}s_1;\ldots;s_{n-2};s_{n-1};s_n \sim \mathbf{c}\{B\}} \qquad \text{if} \qquad fv(s_{n-1}) \cap fv(s_n) = \varnothing.$$

This rule holds since s_{n-1} is $fv(s_{n-1})$-local and s_n is $fv(s_n)$-local, and thus $[\![s_{n-1}; s_n]\!] = [\![s_n; s_{n-1}]\!]$ since $fv(s_{n-1}) \cap fv(s_n) = \varnothing$.

If-statements count as single statements, even if their bodies contain multiple statements.

When invoked as `swap right`, the analogous rule

$$\frac{\{A\}\mathbf{c} \sim s_1;\ldots;s_{n-2};s_n;s_{n-1}\{B\}}{\{A\}\mathbf{c} \sim s_1;\ldots;s_{n-2};s_{n-1};s_n\{\mathbf{c}\}B} \qquad \text{if} \qquad fv(s_{n-1}) \cap fv(s_n) = \varnothing$$

is applied.

Tactic wp

Removes the last statement from the left or right program of a qRHL subgoal and adapts the postcondition accordingly.

More precisely, when invoked as "`wp left.`" or "`wp right.`", it applies the rule

$$\frac{\{A\}s_1;\ldots;s_{n-1} \sim \mathbf{c}\{\mathrm{wp}_1(B,s_n)\}}{\{A\}s_1;\ldots;s_{n-1};s_n \sim \mathbf{c}\{B\}} \quad \text{or} \quad \frac{\{A\}\mathbf{c} \sim s_1;\ldots;s_{n-1}\{\mathrm{wp}_2(B,s_n)\}}{\{A\}\mathbf{c} \sim s_1;\ldots;s_{n-1};s_n\{B\}} \tag{1}$$

respectively. (If-statements count as single statements, even if their bodies contain multiple statements.) Here the wp_1 is the following recursively defined partial function:

$$\begin{aligned}
\mathrm{wp}_1(B,\ \texttt{x <- } e) &:= B\{e_1/\mathbf{x}_1\} \\
\mathrm{wp}_1(B,\ \texttt{x <\$ } e) &:= \mathfrak{Cla}[\texttt{weight } e_1 = 1] \sqcap (\texttt{INF } \mathbf{x}_1 \in \texttt{supp } e_1.\ B) \\
\mathrm{wp}_1(B,\ \texttt{on } \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ apply } e) &:= \mathfrak{Cla}[\texttt{isometry } e_1] \sqcap \big(\bar{e}^* \cdot (B \sqcap \bar{e} \cdot \texttt{top})\big) \\
&\qquad \text{where } \bar{e} := e_1 \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!] \\
\mathrm{wp}_1(B,\ \texttt{x <- measure } \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ with } e) &:= \mathfrak{Cla}[\texttt{mtotal } e_1] \sqcap \big(\texttt{INF } z.\ \big((B\{z/\mathbf{x}_1\} \sqcap \bar{e}) + \texttt{ortho } \bar{e}\big)\big) \\
&\qquad \text{where } \bar{e} := \big((\texttt{mproj } e_1\ z) \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!]\big) \cdot \texttt{top} \\
\mathrm{wp}_1(B,\ \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ <q } e) &:= \mathfrak{Cla}[\texttt{norm } e_1 = 1] \sqcap B \div e_1 \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!] \\
\mathrm{wp}_1(B,\ \texttt{if (}e\texttt{) then c else d}) &:= \big(\mathfrak{Cla}[\neg e_1] + \mathrm{wp}_1(B,\mathbf{c})\big) \sqcap \big(\mathfrak{Cla}[e_1] + \mathrm{wp}_1(B,\mathbf{d})\big) \\
\mathrm{wp}_1(B,\ s_1;\ldots;s_n) &:= \mathrm{wp}_1(\mathrm{wp}_1(\ldots \mathrm{wp}_1(\mathrm{wp}_1(B,s_n),s_{n-1})\ldots,s_2),s_1)
\end{aligned}$$

Here we write e_1 for $\mathrm{idx}_1\, e$ everywhere. Note that the function wp_1 is undefined if the argument contains a `call`-statement. In those cases, the tactic will fail.

The function wp_2 is defined analogously, except that all variables and expressions get index 2 instead of index 1.

The functions wp_1 and wp_2 satisfy $\{\mathrm{wp}_1(B,\mathbf{c})\}\mathbf{c} \sim \mathbf{skip}\{B\}$ and $\{\mathrm{wp}_2(B,\mathbf{c})\}\mathbf{skip} \sim \mathbf{c}\{B\}$, respectively. This can be seen by induction over the structure of $\mathbf{c}$, and using the rules Assign1, Sample1, QApply1,[8] Measure1, QInit1,[9] If1, Conseq, and Seq. From this, the rules in (1) follow with rule Seq.

Note that we call this tactic `wp` like "weakest precondition". However, we stress that we have not actually proven that the precondition returned by wp_1 or wp_2 is indeed the *weakest* precondition. (We have merely tried to make them as weak as possible.)

6 Accompanying Isabelle theories

A proof script for our tool can load an accompanying Isabelle/HOL theory (using the `isabelle` command). In this theory, arbitrary Isabelle/HOL developments are possible. In particular, one can define new types and constants for use in programs (e.g., the encryption scheme in Section 7.1), and one can

[8] Note that rule QApply1 does not contain the term $\mathfrak{Cla}[\texttt{isometry } e_1]$ that $\mathrm{wp}_1(B,\texttt{on } \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ apply } e)$ contains. The reason why $\mathrm{wp}_1(\ldots)$ includes this additional term is that $\texttt{on } \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ apply } e$ actually translates to **apply** $\mathit{mkIso}(e)$ **to** $q_1,\ldots,q_n$ (see footnote 5). Applying rule QApply1 to this program gives the precondition

$$\hat{e}^* \cdot (B \sqcap \hat{e} \cdot \texttt{top}) \quad \text{where} \quad \hat{e} := \mathit{mkIso}(\mathrm{idx}_1\, e) \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!]$$

which is a superset of

$$\mathfrak{Cla}[\texttt{isometry } e_1] \sqcap \bar{e}^* \cdot (B \sqcap \bar{e} \cdot \texttt{top}) \quad \text{where} \quad \bar{e} := e_1 \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!] \text{ and } e_1 := \mathrm{idx}_1\, e.$$

[9] Note that rule QInit1 does not contain the term $\mathfrak{Cla}[\texttt{norm } e_1 = 1]$ that $\mathrm{wp}_1(B,\ \mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ <q } e)$ contains. The reason why $\mathrm{wp}_1(\ldots)$ includes this additional term is that $\mathbf{q}^{(1)},\ldots,\mathbf{q}^{(n)} \texttt{ <q } e$ actually translates to $\mathbf{q}_1,\ldots,\mathbf{q}_n \xleftarrow{\mathfrak{q}} \mathit{mkUnit}(e)$ (see footnote 4). Applying rule QInit1 to this program gives the precondition

$$B \div \hat{e} \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!] \quad \text{where} \quad \hat{e} := \mathit{mkUnit}(\mathrm{idx}_1\, e)$$

which is a superset of

$$\mathfrak{Cla}[\texttt{norm } e_1 = 1] \sqcap B \div e_1 \guillemotright [\![\mathbf{q}_1^{(1)},\ldots,\mathbf{q}_1^{(n)}]\!] \quad \text{where} \quad e_1 := \mathrm{idx}_1\, e.$$

prove arbitrary helper lemmas as long as they do not involve qRHL judgments. (Typically, one will prove lemmas about predicates.) It is beyond the scope of this paper to introduce proofs in Isabelle/HOL, see the tutorial [5] and the reference manual [10] for more information. The theory `QRHL` provides numerous definitions (most of them listed in Section 4) and axioms/lemmas. Many of the lemmas are declared as simplification rules, but some of them are for direct use only. We do not provide a comprehensive list here. To find useful facts, use the `find_theorems` command in Isabelle [10]. Or try the `sledgehammer` command [10] for proving simple lemmas. The following axioms/lemmas correspond to facts proven in this paper (all other axioms/lemmas are well-known or obvious facts):

Isabelle lemma	**Lemma in [9]**
`leq_space_div` [simp]	Lemma 20
`classical_inf` [simp] `classical_sup` [simp] `Cla_plus` [simp] `BINF_Cla` [simp] `classical_ortho` [simp]	Lemma 24
`qeq_collect` `qeq_collect_guarded` [simp]	Lemma 30
`Qeq_mult1` `Qeq_mult2`	Lemma 31
`quantum_eq_unique` [simp]	Lemma 32
`quantum_eq_add_state`	Lemma 33
[simp] means: the lemma is added to the simplifier	

The accompanying theory can also be used to set Isabelle configuration options that then affect our tool's behavior. For example, use

```
declare[[show_types,show_sorts]]
```

in the accompanying theory to add type information to the output of our tool (this affects all Isabelle/HOL formulas printed as part of the subgoals).

Code generation. If all quantum variables involved in a claim about predicates have finite types, the claim will often essentially be a claim about concrete operators and subspaces of fixed dimension. This means that by explicit computation of those operators and subspaces, the claim can be decided. To support this, we use the Isabelle code generation mechanism [2]. This mechanism allows us to provide explicit algorithms for the various operations that occur in formulas. (For example, we might provide a matrix addition algorithm for $A + B$ where $A, B :: (\alpha, \beta)$ `bounded`.) In our case, we give algorithms for most operations on bounded operators and subspaces. (We rely heavily on [7] which implements various algorithms on matrices in Isabelle/HOL.) This allows us to directly evaluate most expressions involving bounded operators and subspaces, as long as the involved types are finite.[10]

Unfortunately, most expressions involving predicates that occur as subgoals in our tool cannot be directly evaluated using this mechanism. This is due to the `lift` (») operation. For example, we might have the claim

$$\texttt{span}\ \{\texttt{EPR}\}\text{»}[\![\mathbf{q}_1, \mathbf{q}_2]\!] \leq \texttt{span}\ \{|00\rangle, |11\rangle\}\text{»}[\![\mathbf{q}_1, \mathbf{q}_2]\!] \tag{2}$$

Here the lhs and rhs are infinite dimensional subspaces (because »$[\![\mathbf{q}_1, \mathbf{q}_2]\!]$ maps a subspace of $\ell^2[\mathbf{q}_1\mathbf{q}_2]$ to a subspace of $\ell^2[V_1^{\mathsf{qu}}V_2^{\mathsf{qu}}]$). Therefore, the lhs and rhs cannot be explicitly computed (at least not using a straighforward representation). Thus, we first need to convert the above expression into the following equivalent finite dimensional one: `span` $\{\texttt{EPR}\} \leq$ `span` $\{|00\rangle, |11\rangle\}$.

In this specific case, this is a special case of the simple rule $A \leq B \implies A\text{»}Q \leq B\text{»}Q$. In general, however, removing the `lift` operations can be nontrivial. The `lift`s can be interspersed with different operations, and they may use different sets of quantum variables, or differently ordered ones. For example, consider

$$\texttt{span}\ \{\texttt{EPR}\}\text{»}[\![\mathbf{q}_1, \mathbf{q}_2]\!] \leq \texttt{span}\ \{|00\rangle, |11\rangle\}\text{»}[\![\mathbf{q}_2, \mathbf{q}_1]\!] \tag{3}$$

(Note the different order $\mathbf{q}_2, \mathbf{q}_1$ on the rhs.) To make this into a finite dimensional expression, we first have to rewrite `span` $\{|00\rangle, |11\rangle\}$»$[\![\mathbf{q}_2, \mathbf{q}_1]\!]$ into $\big(\texttt{comm\_op} \cdot \texttt{span}\ \{|00\rangle, |11\rangle\}\big)$»$[\![\mathbf{q}_1, \mathbf{q}_2]\!]$ (where `comm_op` is

[10] Strictly speaking, besides being finite, the types need to implement the type class `Enum.enum` which means an explicit list of all elements of the type must be provided.

an operator mapping $|x,y\rangle$ to $|y,x\rangle$), and only then can we apply the rule $A \leq B \implies A \triangleright Q \leq B \triangleright Q$ and get

$$\texttt{span}\ \{\texttt{EPR}\} \leq \texttt{comm\_op} \cdot \texttt{span}\ \{|00\rangle, |11\rangle\}. \tag{4}$$

We have automated this process (using a number of simplification rules and custom ML simplification procedures). To perform this conversion, we use the following method:

```
apply (simp add: prepare_for_code)
```

Another problem is that the Isabelle/HOL code generation implements real numbers as fractions. Thus, the code generation fails (aborts) when the expression involves, e.g., square roots. Unfortunately, operators and states such as `hadamard` and `EPR` involve $\sqrt{2}$. We reimplemented the real number code generation in Isabelle/HOL to support real numbers of the form $a + b\sqrt{2}$ for rational a, b, thus these operators and states can be used. However, there is no support so far for other irrational numbers (e.g., $\sqrt{3}$).

To give a complete example, (3) can be shown as follows:

```
lemma
  assumes[simp]: "declared_qvars ⟦q1,q2⟧"
  shows "span {EPR} » ⟦q1,q2⟧ ≤ span {ket (0,0), ket (1,1)} » ⟦q2,q1⟧"
apply (simp add: prepare_for_code)
by eval (* Invokes proof by code evaluation *)
```

This example, the proof of (2), and a few other examples can be found in `Code_Example.thy`.[11] (A remark: the subgoal produced after `apply (simp ...)` in this example is not the same as in (4) but a somewhat more complex one. This is because the simplification procedures do not necessarily find the simplest way of removing the `lift`s.)

7 Examples

7.1 ROR-OT-CPA encryption from PRGs

Our first example proof is the ROR-OT-CPA security of a simple one-time encryption scheme.

The setting. The encryption scheme is defined by

$$\begin{aligned} &\mathrm{enc} : K \times M \to M, \qquad &&\mathrm{enc}(k,m) := G(k) \oplus m \\ &\mathrm{dec} : K \times M \to M, \qquad &&\mathrm{dec}(k,c) := G(k) \oplus c \end{aligned}$$

where $G : K \to M$ is a pseudorandom generator, k is the key, and m is the message (plaintext).

The ROR-OT-CPA security notion says, informally: The adversary cannot distinguish between an encryption of m and an encryption of a random message, even if the adversary itself chooses m. More formally:

Definition 1: ROR-OT-CPA advantage

For a stateful adversary A_1, A_2, let

$$\begin{aligned} \mathrm{Adv}^{A_1A_2}_{\mathrm{ROR}}(\eta) := \Big| &\Pr\big[b = 1 : k \in_\$ K,\ m \leftarrow A_1(),\ c := \mathrm{enc}(k,m),\ b \leftarrow A_2(c)\big] \\ &- \Pr\big[b = 1 : k \in_\$ K,\ m \leftarrow A_1(),\ r \in_\$ M,\ c := \mathrm{enc}(k,r),\ b \leftarrow A_2(c)\big]\Big| \end{aligned}$$

where $\in_\$$ means uniformly random choice, and the notation $\Pr[e : G]$ denotes the probability that e holds after running the instructions in G, and η is a security parameter (on which A_1, A_2, G, K, M implicitly depend).

We call $\mathrm{Adv}^{A_1A_2}_{\mathrm{ROR}}$ the *ROR-OT-CPA advantage* of A_1, A_2.

With this definition, we can then, for example, define ROR-OT-CPA security of enc as "for any quantum-polynomial-time A_1, A_2, $\mathrm{Adv}^{A_1A_2}_{\mathrm{ROR}}$ is negligible." This is what is called asymptotic security. We will instead follow the concrete security approach where we explicitly derive bounds for $\mathrm{Adv}^{A_1A_2}_{\mathrm{ROR}}$.

[11] Bundled with the tool, and also directly available at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/Code_Example.thy`.

Analogously, we define pseudorandomness of $G : K \to M$ by defining the *PRG advantage* of G:

Definition 2: PRG advantage

For an adversary A, let

$$\mathrm{Adv}^A_{\mathrm{PRG}}(\eta) := \Big|\Pr\big[b = 1 : s \in_\$ K,\, r := G(s),\, b \leftarrow A(r)\big] - \Pr\big[b = 1 : r \in_\$ M,\, b \leftarrow A(r)\big]\Big|.$$

Again, we can define pseudorandomness of G by requiring that $\mathrm{Adv}^A_{\mathrm{PRG}}$ is negligible for all quantum-polynomial-time A, or reason about concrete advantages.

What we want to show is the following well-known fact: "If G is pseudorandom, then enc is ROR-OT-CPA." In the concrete security setting, we can state this more precisely:

Lemma 3: Concrete ROR-OT-CPA security of enc

For any A_1, A_2, there exists a B such that:
(i) $\mathrm{Time}(B) \le \mathrm{Time}(A_1) + \mathrm{Time}(A_2) + O(\log \eta)$.
(ii) $\mathrm{Adv}^{A_1,A_2}_{\mathrm{ROR}}(\eta) \le \mathrm{Adv}^B_{\mathrm{PRG}}(\eta)$.
Here $\mathrm{Time}(A)$ refers to the worst-case runtime of A, and we assume that elementary operations (e.g., $\oplus$) on K and M take time $O(\log \eta)$.

It is immediate that this also implies asymptotic ROR-OT-CPA security.

In our tool, we will almost show Lemma 3. Specifically, we will show property (ii), but we will not show (i) (because our tool does not have the concept of the runtime of an algorithm). Instead, we explicitly specify B and leave it to the user to check that B indeed satisfies (i). This is the state of the art and is done in the same way, in, e.g., EasyCrypt and CryptHOL. Explicit reasoning about runtime is left as future work.

In addition, we will leave the security parameter η implicit. This means that our proof is for fixed η, but since it holds for any η, the case of variable η is implied.

Specification in Isabelle. The first step is to encode the encryption scheme itself. Since this involves the definition of types (for keys and messages) and logical constants (enc and G), it needs to be done in an accompanying Isabelle theory `PrgEnc.thy`.[12]

In this theory, we first declare the types `key` and `msg` as abstract (i.e., unspecified) types:

```
typedecl key
typedecl msg
```

Furthermore, we need to declare those types as finite (otherwise uniform sampling of keys/messages is not well-defined). We use Isabelle's type class mechanism for this:

```
instantiation key :: finite [...]
instantiation msg :: finite [...]
```

Since we cannot prove that those types are finite (they are declared abstractly, after all), the proof obligations from those `instantiation` commands are discharged with `sorry` (which tells Isabelle to simply believe the claim).

We also need to declare that there is an operation + (the XOR operation) on `msg` that is a group operation, and that satisfies the rule $a + a = 0$. This is done by declaring `msg` as an additive group via the type class mechanism and adding an axiom that specifies the cancellation property:

```
instantiation msg :: group_add [...]
axiomatization where xor_cancel[simp]: "a+a=0" for a::msg
```

Now we can declare the PRG G and the encryption function enc. Since G is just an unspecified function, all we need to do is to declare an uninterpreted constant with the right type. And enc can be explicitly defined:

```
axiomatization G :: "key ⇒ msg"
definition enc :: "key * msg ⇒ msg"
  where [simp]: "enc = (λ(k,x). G(k)+x)"
```

[12] The full theory file is bunded with the tool, and also directly available at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/PrgEnc.thy`.

Games from Definition 1

```
program rorcpa0 := {
  k <$ uniform UNIV;
  call A1;
  c <- enc(k,m);
  call A2;
}.

program rorcpa1 := {
  k <$ uniform UNIV;
  call A1;
  r <$ uniform UNIV;
  c <- enc(k,r);
  call A2;
}.
```

Games from Definition 2

```
program prg0 := {
  s <$ uniform UNIV;
  r <- G(s);
  call B;
}.

program prg1 := {
  r <$ uniform UNIV;
  call B;
}.
```

Figure 2: Specification of games in `prg-enc-rorcpa.qrhl`.

In addition, we declare an prove some simple simplification rules for XOR that will be used in the proof (`my_simp`, `mysimp2`, `aux_bij`).

Specification in our tool. We now proceed to the specifications that are done in our tool directly. We show only excerpts, the full file is `prg-enc-rorcpa.qrhl`.[13] We first specify the games from Definition 1 and Definition 2. Consider the lhs game from Definition 1. At first, it seems like we have a problem here. The description of the game requires A_1, A_2 to be algorithms that take arguments and return values, i.e., procedures. But our language for programs does not support procedures. Fortunately, there is a simple workaround. We set aside a few global variables (`m,c,r,b`) explicitly for storing inputs and outputs of the adversary. So, for example, $b \leftarrow A_2(c)$ can be performed by declaring b, c as variables accessible to A_2, and then simply calling A_2 without arguments in our program. The former is achieved by the following commands:

```
adversary A1 vars m,cglobA,qglobA.
adversary A2 vars c,b,cglobA,qglobA.
```

(Here `cglobA` and `qglobA` are quantum variables that model the internal classical and quantum state of A_1, A_2.) And for calling the adversary A_2, we have the syntax `call A2;`. The resulting program code is given in Figure 2. Note that `UNIV` is the set of all values (of a given type), so `uniform UNIV` samples uniformly from all keys or messages, respectively.

While A_1 and A_2 are declared as unspecified adversaries, we need to specify B explicitly. (Recall that we wanted to give an explict B so that the user can verify Lemma 3 (i).) In our case, the adversary B is quite simple:

```
program B := { call A1; c <- r+m; call A2; }.
```

It is easy to see that (assuming a suitable formalization of runtime) the overhead of B is only $O(\log \eta)$.

The proof. The proof proceeds by first proving two facts as lemmas:

```
lemma rorcpa0_prg0: Pr[b=1:rorcpa0(rho)] = Pr[b=1:prg0(rho)].
lemma rorcpa1_prg1: Pr[b=1:rorcpa1(rho)] = Pr[b=1:prg1(rho)].
```

Here `rho` is an ambient variable of type `program_state`, so the lemmas hold for any initial state `rho`. Recall that `Pr[b=1:G(rho)]` refers to the probability that `b` = 1 after `G`.

The proofs of both lemmas have similar form. In both cases, we first transform the claim into a qRHL judgment using the tactic `byqrhl` We inline the definitions of `rorcpa0`, `prg0`, and `B` using the `inline` tactic. Trailing assignments are removed with `wp left` or `wp right` when they occur. Ambient subgoals are proven using the `simp` tactic, possibly giving some of the auxiliary lemmas from `PrgEnc.thy` as hints. And for subgoals of the form $\{\ldots\}\ldots;$`call` $A \sim \ldots;$`call` $A\{\ldots\}$,

[13] Bundled with the tool, and also directly available at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/prg-enc-rorcpa.qrhl`.

we use the `equal` tactic to remove the last statement. We use the `swap` tactic to swap two statements where needed to make matching `call`-statements occur together. Similarly, for subgoals `{...}...;k <$ uniform UNIV` $\sim$ `...;s <$ uniform UNIV{...}`, we use the `rnd` tactic. In the proof of lemma `rorcpa0_prg0`, we will need k and s to be sampled identically, so the basic form `rnd.` of the tactic is sufficient. In `rorcpa1_prg1` we encounter a more interesting case: We have the subgoal

```
{...}...; r <$ uniform UNIV; ~ ...; r <$ uniform UNIV;{ Cla[G k1 + r1 = r2 + m2
                                  /\ b1 = b2 /\ cglobA1 = cglobA2] |^| [[qglobA1]] == q [[qglobA2]] }
```

At the first glance, it would seem that the right thing to do is to sample `r1` and `r2` identically by applying `rnd.` However, if `r1` = `r2`, then the part `G k1 + r1 = r2 + m2` of the postcondition will not be satisfied. Instead, we want to pick `r1` and `r2` such that their XOR is `r + G k1 + m2`. This can be achieved by the extended form of the `rnd` tactic that provides a witness for the joint distribution of `r1` and `r2`:

```
rnd r,r <- map_distr (λr. (r,r + G k1 + m2)) (uniform UNIV).
```

This means `r` is picked uniformly, and `r1` is `r`, and `r2` is `r + G k1 + m2` which makes the postcondition true.

After having shown lemmas `rorcpa0_prg0` and `rorcpa1_prg1`, we can show Lemma 3 in the following form:

```
lemma final: abs (Pr[b=1:rorcpa0(rho)] - Pr[b=1:rorcpa1(rho)])
           = abs (Pr[b=1:prg0(rho)] - Pr[b=1:prg1(rho)]).
```

This fact follows immediately (using the Isabelle simplifier) from the lemmas `rorcpa0_prg0` and `rorcpa1_prg1`, so we can show it using `simp ! rorcpa0_prg0 rorcpa1_prg1.`

7.2 IND-OT-CPA encryption from PRGs

The second example is the IND-OT-CPA security of the encryption scheme enc from Section 7.1. We give this second example to show that security proofs that contain more than one reduction step do not pose a problem. (The ROR-OT-CPA proof from Section 7.1 was a single reduction step to the PRG security of G.) We only describe the differences to the proof from Section 7.1.

The setting. The IND-OT-CPA security notion says, informally: The adversary cannot distinguish between an encryption of m_1 or m_2, even if the adversary chooses m_1 and m_2 itself. More formally:

Definition 4: IND-OT-CPA advantage

For a stateful adversary A_1, A_2, let

$$\mathrm{Adv}^{A_1A_2}_{\mathrm{IND}}(\eta) := \Big|\Pr\big[b = 1 : k \in_\$ K,\ (m_1, m_2) \leftarrow A_1(),\ c := \mathrm{enc}(k, m_1),\ b \leftarrow A_2(c)\big] - \Pr\big[b = 1 : k \in_\$ K,\ (m_1, m_2) \leftarrow A_1(),\ c := \mathrm{enc}(k, m_2),\ b \leftarrow A_2(c)\big]\Big|$$

We call $\mathrm{Adv}^{A_1A_2}_{\mathrm{IND}}$ the *IND-OT-CPA advantage* of A_1, A_2.

What we want to show is the following well-known fact: "If G is pseudorandom, then enc is IND-OT-CPA." In the concrete security setting, we can state this more precisely:

Lemma 5: Concrete IND-OT-CPA security of enc

For any A_1, A_2, there exist B_1, B_2 such that:
(i) $\mathrm{Time}(B_i) \le \mathrm{Time}(A_1) + \mathrm{Time}(A_2) + O(\log \eta)$ for $i = 1, 2$.
(ii) $\mathrm{Adv}^{A_1,A_2}_{\mathrm{IND}}(\eta) \le \mathrm{Adv}^{B_1}_{\mathrm{PRG}}(\eta) + \mathrm{Adv}^{B_2}_{\mathrm{PRG}}(\eta)$.

As before, we will not show (i) in the tool but instead define B_1 and B_2 explicitly, leaving the runtime analysis to the user.

Games from Definition 4

```
adversary A1 vars m1,m2,cglobA,qglobA.
adversary A2 vars c,b,cglobA,qglobA.

program indcpa0 := {
  k <$ uniform UNIV;
  call A1;
  c <- enc(k,m1);
  call A2;
}.

program indcpa1 := {
  k <$ uniform UNIV;
  call A1;
  c <- enc(k,m2);
  call A2;
}.
```

Games from Definition 2

```
program prg0B1 := {
  s <$ uniform UNIV;
  r <- G(s);
  call B1; }.

program prg1B1 := {
  r <$ uniform UNIV;
  call B1; }.

program prg0B2 := {
  s <$ uniform UNIV;
  r <- G(s);
  call B2; }.

program prg1B2 := {
  r <$ uniform UNIV;
  call B2; }.
```

Figure 3: Specification of games in `prg-enc-indcpa.qrhl`.

Specification. The specification of the encryption scheme enc and the PRG G is unchanged. That is, we use the same accompanying theory `PrgEnc.thy` as in Section 7.1.

In our tool,[14] we have to describe the two IND-OT-CPA games from Definition 4 (`indcpa0` and `indcpa1` in Figure 3), as well as the two PRG games from Definition 2. For the latter, there is a minor issue: Since we have two reductions to the security of G, we need to invoke the security of G twice, once for the adversary B_1, and once for the adversary B_2. Since our tool does not have a module system that would allow us to generically instantiate the same game with different adversaries (e.g., EasyCrypt's module system allows us to specify the games with a module parameter that is then instantiated with an adversary module), we need to write down the games from Definition 2 twice, once for adversary B_1 (`prg0B1` and `prg1B1` in Figure 3) and once for adversary B_2 (`prg0B1` and `prg1B1`).

And, of course, we need to explicitly specify the adversaries B_1 and B_2:

```
program B1 := { call A1; c <- r+m1; call A2; }.
program B2 := { call A1; c <- r+m2; call A2; }.
```

It is easy to see that they satisfy the runtime conditions in Lemma 5 (i).

The proof. We use the following sequence of games:

$$\boxed{\texttt{indcpa0}} \overset{=}{\longleftrightarrow} \boxed{\texttt{prg0B1}} \overset{\mathrm{Adv}^{B_1}_{\mathrm{PRG}}}{\longleftrightarrow} \boxed{\texttt{prg1B1}} \overset{=}{\longleftrightarrow} \boxed{\texttt{prg1B2}} \overset{\mathrm{Adv}^{B_2}_{\mathrm{PRG}}}{\longleftrightarrow} \boxed{\texttt{prg0B2}} \overset{=}{\longleftrightarrow} \boxed{\texttt{indcpa1}}$$

Here $\overset{=}{\longleftrightarrow}$ means that we show that the probability of $b = 1$ is the same in the two games. And $\overset{\mathrm{Adv}^{B_i}_{\mathrm{PRG}}}{\longleftrightarrow}$ means that the difference of $\Pr[b = 1]$ is $\mathrm{Adv}^{B_i}_{\mathrm{PRG}}$ (we do not need to prove those arrows, since that difference between those games is $\mathrm{Adv}^{B_i}_{\mathrm{PRG}}$ by definition).

The three $\overset{=}{\longleftrightarrow}$ are shown in the following lemmas:

```
lemma indcpa0_prg0B1: Pr[b=1:indcpa0(rho)] = Pr[b=1:prg0B1(rho)].
lemma prg1B1_prg1B21: Pr[b=1:prg1B1(rho)]  = Pr[b=1:prg1B2(rho)].
lemma indcpa1_prg0B2: Pr[b=1:indcpa1(rho)] = Pr[b=1:prg0B2(rho)].
```

The proofs of these lemmas are similar to the ones in Section 7.1.

From these three lemmas we immediately get the final result (which encodes Lemma 5 (ii)):

```
lemma final: abs( Pr[b=1:indcpa0(rho)] - Pr[b=1:indcpa1(rho)] ) <=
             abs( Pr[b=1:prg0B1(rho)]  - Pr[b=1:prg1B1(rho)] ) +
             abs( Pr[b=1:prg0B2(rho)]  - Pr[b=1:prg1B2(rho)] ).
```

This can be proven immediately using the tactic `simp ! indcpa0_prg0B1 indcpa1_prg0B2 prg1B1_prg1B21.`

[14] File `prg-enc-indcpa.qrhl`, bundled with the tool, and also directly available here: `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/prg-enc-indcpa.qrhl`.

7.3 Quantum equality

In the file `equality.qrhl`[15] we give a simple example involving reasoning about quantum equality. We show

$$\{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\}\texttt{prog1} \sim \texttt{prog2}\{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\} \tag{5}$$

for the following programs:

```
program prog1 := {
  b <$ uniform UNIV;
  if (b=1) then on q apply hadamard;
           else skip;
}.
```

```
program prog2 := {
  on q apply hadamard;
  b <$ uniform UNIV;
  on q apply (if b=1
     then hadamard else idOp); }.
```

The first program `prog1` picks a random bit b and applies the Hadamard operation H to $\mathbf{q}$ iff $b = 1$. The second program `prog2` additionally first applies H, then picks b, and then applies H iff $b = 1$. Since $H^2 = id$, in both programs H is applied to $\mathbf{q}$ with probability $\frac{1}{2}$, so we expect them to have the same effect on $\mathbf{q}$. This is what (5) expresses.

There are two important differences between `prog1` and `prog2`. First, `prog2` performs an additional application of H which means that the $b = 1$ case of `prog2` corresponds to the $b = 0$ case in `prog1` and vice versa. And secondly, we have written the conditional application of H differently. In `prog1`, if $b = 1$, H is applied, otherwise nothing is done. In contrast, in `prog2`, there is always an application on $\mathbf{q}$, but the operator that is applied is computed using the expression `if b=1 then hadamard else idOp` which evaluates to H or to the identity. In other words, in `prog1`, we use a language-level conditional and perform an actual branching. While in `prog2`, no branching occurs, and the conditional is encoded in the computation of the unitary that is applied. Of course, this should not make a difference, but we formulated the two programs differently to demonstrate that our logic can handle both approaches gracefully.

We will formalize two proofs. The first is a bit longer, and explicitly states the invariants and case distinctions that are made. This makes the proof more instructive. The second proof makes is as terse as possible, simply applying tactics to remove statements from the end of the programs, and relying on the simplifier to remove the final, lengthy, verification condition.

The "instructive" proof. We start with the qRHL subgoal

$$\{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\}\texttt{call prog1;} \sim \texttt{call prog2;}\{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\}$$

and use the tactic `inline` to inline the code of both programs. Then we use `seq 0 1:` I_1 with I_1 := `quantum_equality_full idOp ⟦q1⟧ hadamard ⟦q2⟧` to split off the first statement of the right program. That is, we claim that after executing the first statement of the right program (an application of Hadamard H on $\mathbf{q}$), the precondition $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$ is transformed into $id\,\mathbf{q}_1 \equiv_{\mathsf{quant}} H\mathbf{q}_2$. Intuitively, this is what we expect, because if originally $\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2$, and the new $\mathbf{q}_2$ is the result of applying H to $\mathbf{q}_2$, then the new $\mathbf{q}_2$ should equal $\mathbf{q}_1$ if we apply another H to it. The resulting subgoal can be solved easily using `wp right. skip. simp.`

We are left with the new goal

$$\{I_1\}\ \texttt{b <\$ uniform UNIV; if (b=1) then on q apply hadamard; else skip;}$$
$$\sim \texttt{b <\$ uniform UNIV; on q apply (if b=1 then hadamard else idOp);}\ \{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\}$$

We then claim that the sampling of b on both sides leads to $b_1 \neq b_2$. That is, we use the tactic `seq 1 1:` I_2 with I_2 := `quantum_equality_full idOp ⟦q1⟧ hadamard ⟦q2⟧ ⊓ Cla[b1≠b2].` to split off the two samplings into a separate qRHL judgement. That judgement can be solved using the `rnd` tactic. Since we want $b_1 \neq b_2$ to hold, we cannot use the simple form of `rnd`, but instead we use `rnd b,b <- map_distr (λb. (b,b+1)) (uniform UNIV)` to tell the tool to sample b_1 and b_2 so that they will always be inequal. (Note: $b + 1$ is the negation of the bit b since $+$ is XOR on bits.) We use `skip. simp!` to discharge the remainder of this subgoal.

Now, we are left with the subgoal

[15] Bundled with the tool, and also available directly at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/equality.qrhl`.

```
program teleport := {
  A,B <q EPR;
  on C,A apply CNOT;
  on C apply hadamard;
  a <- measure A with computational_basis;
  c <- measure C with computational_basis;
  if (a=1) then on B apply pauliX;
           else skip;
  if (c=1) then on B apply pauliZ;
           else skip;  }.
```

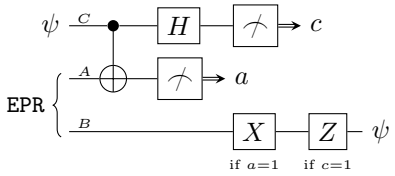

Figure 4: Quantum teleportation as a program and as a circuit.

$$\{I_2\}\ \texttt{if (b=1) then on q apply hadamard; else skip;} \sim \texttt{on q apply (if b=1 then hadamard else idOp);}\ \{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\} \quad (6)$$

Note that I_2 contains the program variables b_1, b_2 upon which further branching depends. To be able to make a case distinction over their values, we need to be able to refer to their values in the ambient logic. To this end, we apply the tactic `case z := b1`. This adds $\mathfrak{Cla}[\texttt{b1 = z}]$ to the precondition where z is an ambient variable. (That means that we can treat z as a fixed value and make a case distinction over its value.) The case distinction itself is done via `casesplit z=0.` This will create two new subgoals, one with the additional assumption (in the ambient logic, not in the precondition) that $z = 0$, and one that $z \neq 0$. The rest of the subgoal is still as in (6).

To finish the first subgoal, we apply `wp left. wp right.` which removes the remaining statements and changes the postcondition accordingly. Then `skip. simp.` solves the subgoal. The $z \neq 0$ subgoal is solved analogously.

The "terse" proof. As it turns out, the previous proof is much more verbose than needed. Instead of explicitly using `seq`, `case`, and `casesplit` to decompose the proof into understandable subgoals, we can use the "straightforward" approach and simply remove statement by statement from the end of the programs, and leave it to the simplifier to prove the resulting statement. That is, we use `wp left. wp right` to remove the conditional applications of H, then we use `rnd b,b <- map_distr (λb. (b,b+1)) (uniform UNIV)` to remove the two samplings (in a way that ensures $b_1 \neq b_2$). We use `wp right` to remove the remaining application of H in the first line of the right program, and then apply `skip.` We get a lengthy and hardly readable verification condition, but fortunately, it can be discharged by an application of `simp.`

Why did we need the more complex approach in the first proof? In this simple example, we did not. However, in more complex cases, breaking the proof down in individual cases, and simplifying intermediate pre- and postconditions may make it easier for the simplifier (if the overall goal is too complex to be solved in one go), and it may help the user to debug the proof. (For example, to figure out the right witness to be used in the `rnd b,b <- ...` tactic, it helps to have a readable pre- and postcondition. And case distinctions help us to distinguish in which case a problem arises and to narrow down what it is.

7.4 Quantum teleportation

The final example is the analysis of quantum teleportation [1]. Quantum teleportation is a quantum protocol that allows us to move a qubit from a quantum register C to a quantum register B with only classical communication between the system containing C and the system containing B (assuming a shared initial state). The program `teleport` that describes the teleportation process is shown in Figure 4. We will show the following fact:

$$\{C_1 \equiv_{\mathsf{quant}} A_2\}\texttt{teleport} \sim \textbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\} \quad (7)$$

That is, we show that if C_1 contains a qubit that is equal to A_2, then after teleporting C_1 to B_1, B_1 will be equal to A_2 as expected.

As with the example from Section 7.3, we formalize two proofs of (7), an "instructive" one with explictly stated intermediate invariants and case distinctions, and a "terse" one that simply applies `wp` as often as needed and relies on Isabelle to decide the final verification condition.

This example serves both as an illustration that we can analyze protocols that make use of non-trivial quantum effects (as opposed to the examples in Section 7.1 and Section 7.2 which simply maintained equality between two quantum states without ever performing any explicit operations on it), and as a further example on how to use the quantum equality.

The "instructive" proof. This proof is formalized in `teleport.qrhl`.[16] The initial subgoal is (7). We use the tactic `inline teleport` to inline the definition of `teleport`. First, we reason about the first instruction in `teleport`, the initialization of A, B with an EPR state (`A,B <q EPR`). We claim that after that step, the invariant $I_1 := (C_1 \equiv_{\mathsf{quant}} A_2) \sqcap (\texttt{span}\{\texttt{EPR}\} \guillemotright [\![A_1, B_1]\!])$ holds. Intuitively, this is what we expect, since after initializing A, B with `EPR` on the left side, their state with will be in span $\{$`EPR`$\}$. We formalize this with the tactic `seq 1 0:` I_1, and the resulting subgoal can be proven directly using `wp left. skip. simp.`

Then we rewrite the precondition I_1 into

$$I_2 := (\texttt{quantum\_equality\_full idOp } [\![C_1, A_1, B_1]\!] \ (\texttt{addState EPR}) \ [\![A_2]\!])$$

using tactic `conseq pre:` I_2`.` We get a new subgoal $I_1 \leq I_2$ which can be proven using `simp quantum_eq_add_state.` (`quantum_eq_add_state` is the Isabelle formulation of Lemma 33 in [9]). Intuitively, I_2 states that after the initialization, $C_1A_1B_1$ are in the same state as A_2 would be if we were to add the state `EPR` to it.

We now have the subgoal

$$\{I_2\}\mathbf{c}_1 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

where $\mathbf{c}_1$ is `teleport` without the first line.

We now show that the next two lines (applying CNOT and Hadamard) lead to the following invariant:

$$I_3 := \Big(\texttt{quantum\_equality\_full idOp } [\![C_1, A_1, B_1]\!] \\ \big((\texttt{hadamard} \otimes \texttt{idOp}) \cdot \texttt{assoc\_op}^* \cdot (\texttt{CNOT} \otimes \texttt{idOp}) \cdot \texttt{assoc\_op} \cdot \texttt{addState EPR}\big) \ [\![A_2]\!]\Big)$$

In other words, we claim that after those two lines, the quantum registers $C_1A_1B_1$ will contain the state that A_2 would contain if we added the state `EPR` to it, and then applied CNOT on the first two and Hadamard on the first register. What are the unexpected additional operations `assoc_op` and `assoc_op`*? These are needed due to the fact that in Isabelle/HOL, $(\alpha \times \beta) \times \gamma$ and $\alpha \times (\beta \times \gamma)$ are not the same type, although in handwritten mathematics, one usually identifies those types. For example `addState EPR` is an operator from $\ell^2(\texttt{bit})$ to $\ell^2(\texttt{bit} \times (\texttt{bit} \times \texttt{bit}))$. And $\texttt{CNOT} \otimes \texttt{idOp}$ is an operator on $\ell^2((\texttt{bit} \times \texttt{bit}) \times \texttt{bit})$. So we cannot multiply those operators (a type error would be raised by Isabelle and by our tool). Instead, we need to apply `assoc_op` in between, which is the canonical isomorphism between $\ell^2(\texttt{bit} \times (\texttt{bit} \times \texttt{bit}))$ to $\ell^2((\texttt{bit} \times \texttt{bit}) \times \texttt{bit})$. (If we identify $(\alpha \times \beta) \times \gamma$ and $\alpha \times (\beta \times \gamma)$, then `assoc_op` is the identity.) Similarly, `assoc_op`* is the canonical isomorphism in the opposite direction.

In the tool, claiming that the new invariant after the CNOT and the Hadamard is I_3 is done via the tactic `seq 2 0:` I_3`.` To prove the new subgoal resulting from `seq`, we apply `wp left. wp left. skip. simp.` This leaves us with an ambient subgoal relating quantum predicates. Unfortunately, the `simp` tactic is not able to solve this subgoal. Therefore we outsourced this subgoal to Isabelle/HOL. Namely, we copy-and-pasted the subgoal into the accompanying theory `Teleport.thy`,[17] That is, we proved a lemma of the form

```
lemma teleport_goal1:
assumes[simp]: "declared_qvars ⟦A1,B1,C1,A2⟧"
shows "..."
```

where `...` is the copy-and-pasted subgoal. Note the assumption `"declared_qvars ⟦A1,B1,C1,A2⟧"`. This one basically tells Isabelle that `A1,B1,C2,A2` can be treated as distinct quantum variables. (Because logically, free variables in an Isabelle lemma do not have to refer to different entities.) With

[16] Bundled with the tool, and also directly available at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/teleport.qrhl`.

[17] Bundled with the tool, and also directly available at `https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/Teleport.thy`.

this assumption added to the simplifier (using `[simp]`), simplification rules that reason about quantum variables will work correctly. The lemma is proven by stating two intermediate simple facts and then running the simplifier with a collection of facts from `QRHL.thy`. We omit the details. Once we have shown `telepost_goal1` in Isabelle, we can use it in our tool. Namely, to prove the subgoal, we use the tactic `rule telepost_goal1` in our tool. This leaves us with one new subgoal (corresponding to the `"declared_qvars ⟦A1,B1,C1,A2⟧"` assumption of the lemma) which can be discharged by `simp`.

The goal is now:

$$\{I_3\}\mathbf{c}_3 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

where $\mathbf{c}_3$ refers to `teleport` without the first three lines.

Next we analyze the effect of the first measurement. If the outcome of the measurement is a_1, then this means that the state of A_1 is projected onto $|a_1\rangle_{A_1}$. So, after the measurement, the predicate $I_4 := \texttt{Proj (span \{ket } a_1\texttt{\})»⟦}A_1\texttt{⟧} \cdot I_3$ should be satisfied. We express this using the tactic `seq 1 0:` I_4`.` To prove the resulting subgoal, we apply the tactics `wp left. simp. skip. simp` as usual. This leaves us with an ambient subgoal of roughly the following form:

$$\forall x.\ I_3 \leq \texttt{Proj (span \{ket } x\texttt{\})»⟦}A_1\texttt{⟧} \cdot I_3 \sqcap \texttt{span \{ket } x\texttt{\}»⟦}A_1\texttt{⟧} + \texttt{ortho (span \{ket } x\texttt{\}»⟦}A_1\texttt{⟧})$$

We remove the all-quantifier using tactic `fix a'`. Then the fact can be shown using tactic `rule move_plus_meas_rule`,[18] followed by simplification.

We are now left with the goal

$$\{I_4\}\mathbf{c}_4 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

where $\mathbf{c}_4$ is `teleport` without the first four lines.

In order to be able to refer to the value of a_1 in the ambient logic, we apply the tactic `case a'`, this changes the subgoal into

$$\{\mathfrak{Cla}[a_1 = a'] \sqcap I_4\}\mathbf{c}_4 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

We now analyze the effect of the second measurement. If the outcome of the measurement is c_1, then this means that the state of C_1 is projected onto $|c_1\rangle_{C_1}$. So, after the measurement, the predicate $I_5 := \mathfrak{Cla}[a_1 = a'] \sqcap \texttt{Proj (span \{ket } c_1\texttt{\})»⟦}C_1\texttt{⟧} \cdot I_4$ holds. This step is similar to the previous one (`seq 1 0:` I_5 etc.), we omit the details. We again use tactic `case c'` to be able to refer to c_1 in the ambient logic. We have the following goal:

$$\{\mathfrak{Cla}[c_1 = c'] \sqcap I_5\}\mathbf{c}_5 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

where $\mathbf{c}_5$ is `teleport` without the first five lines.

Now we will do a case distinction over the four different possibilities for a', c'. We get the first case using the tactics `casesplit a'=0. casesplit c'=0.` The current subgoal now has the assumptions $a' = 0$ and $c' = 0$. Using these assumptions, we can rewrite the precondition into

$$I_6 := \mathfrak{Cla}[a_1 = 0 \wedge c_1 = 0] \sqcap \underbrace{\texttt{Proj (span \{ket 0\})»⟦}C_1\texttt{⟧} \cdot \texttt{Proj (span \{ket 0\})»⟦}A_1\texttt{⟧} \cdot I_3}_{=:I_7}$$

using `conseq pre:` I_6`.` Besides minor reordering of terms, we basically just substituted $a' := 0$ and $c' := 0$ (which is justified by the assumptions), so the resulting subgoal can be solved directly by `simp!.` The goal is then:

$$\{I_6\}\mathbf{c}_5 \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

Now we analyze the remaining two lines of `teleport`, namely the conditionally applied unitaries `pauliX` and `pauliZ`. In the case $a_1 = 0, c_1 = 0$, they will not be applied, so after the last two lines, the predicate I_7 is still satisfied. (In the other three cases, additionally `pauliZ»⟦`B_1`⟧` and/or `pauliX»⟦`B_1`⟧` would be multiplied to I_7.) We show this using `seq 2 0:` I_7`. wp left. wp left. skip. simp!.`

We finally have the subgoal

$$\{I_7\}\mathbf{skip} \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\}$$

This is transformed into $I_7 \leq (B_1 \equiv_{\mathsf{quant}} A_2)$ by tactic `skip`. What does this inequality say? It says that if we have a state on $C_1A_1B_1$ that is equal to A_2 after adding `EPR` and applying CNOT and Hadamard, and then we apply projections onto $|0\rangle_{A_1}$ and $|0\rangle_{C_1}$ to the state, then that state satisfies $B_1 \equiv_{\mathsf{quant}} A_2$.

[18] The lemma `move_plus_meas_rule` says `(Proj` C`)»`$Q \cdot A \leq B \implies A \leq (B \sqcap C\texttt{»}Q) + (\texttt{ortho } C\texttt{»}Q$ and is useful for simplifying inequalities between predicates arising from `wp` applied to a measurement.

Showing this inequality is the core of the actual proof that teleportation works. We show this inequality by explicit computation of the involved operators and subspaces. We use the code generation mechanism of Isabelle for this explicit computation. That is, we copy-and-paste the subgoal into the accompanying theory `Teleport.thy` as a lemma.

```
lemma teleport_goal2_a0c0:
  assumes[simp]: "declared_qvars ⟦A1,B1,C1,A2⟧"
  shows "I₇ ≤ (B₁ ≡quant A₂)"
    apply (simp add: prepare_for_code) by eval
```

(See Section 6 for an explanation of `prepare_for_code` and `eval`.) With this lemma in the accompanying theory, we can solve the goal in our tool using `rule teleport_goal2_a0c0. simp!.`

The other three cases for a', c' are solved analogously.

The "terse" proof. The proof described above shows the predicates that hold after each step of the teleportation program. However, a much shorter (and less explicit) proof is possible, too. This proof is given in `teleport-terse.qrhl`.[19] The definition of the program `teleport` is the same as before (see Figure 4). To prove the goal (7), we unfolding the definition of `teleport` using `inline teleport`, then apply the tactic `wp left` seven times to get a goal of the form $\{\ldots\}\mathbf{skip} \sim \mathbf{skip}\{\ldots\}$, and the apply `skip`. We now get a lengthy inequality between predicates as the remaining goal. While this inequality is hardly readable (it is an 814 character string), we can just copy-and-paste it into the accompanying theory `Teleport_Terse.thy`[20] as a lemma `teleport_terse` and prove it using Isabelle's code generation (see Section 6). With that lemma, the goal is proven with the tactics `rule teleport_terse. simp!.`

References

[1] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels". In: *Phys. Rev. Lett.* 70 (13 1993), pp. 1895–1899. DOI: 10.1103/PhysRevLett.70.1895.

[2] Florian Haftmann. *Code generation from Isabelle/HOL theories*. https://isabelle.in.tum.de/dist/Isabelle2017/doc/codegen.pdf.

[3] Lars Hupel, Frank S. Thomas, and Alexandre Archambault. *larsrh/libisabelle: libisabelle 0.9.2*. Oct. 2017. DOI: 10.5281/zenodo.1012471.

[4] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[5] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*. https://isabelle.in.tum.de/dist/Isabelle2017/doc/prog-prove.pdf. 2017.

[6] The PG dev team. *Proof General (A generic Emacs interface for proof assistants)*. https://proofgeneral.github.io/. Accessed 2017-01-02.

[7] René Thiemann and Akihisa Yamada. "Matrices, Jordan Normal Forms, and Spectral Radius Theory". In: *Archive of Formal Proofs* (Aug. 2015). http://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development. ISSN: 2150-914x.

[8] Dominique Unruh. *dominique-unruh/qrhl-tool: Prototype proof assistant for qRHL*. GitHub. 2018. URL: https://github.com/dominique-unruh/qrhl-tool.

[9] Dominique Unruh. *Quantum Relational Hoare Logic*. arXiv:1802.03188v1 [quant-ph]. 2018.

[10] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/dist/Isabelle2017/doc/isar-ref.pdf. 2017.

Symbol index

$\mathbf{D}(B)$	Distributions on B

[19] Bundled with the tool, and also directly available at https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/teleport-terse.qrhl.

[20] Bundled with the tool, and also directly available at https://raw.githubusercontent.com/dominique-unruh/qrhl-tool/master/Teleport_Terse.thy.

$\mathbf{D}[V]$	$\mathbf{D}(\mathsf{Type}_V^{\mathsf{set}})$ – Distributions on $\mathsf{Type}_V^{\mathsf{set}}$	
$\mathbf{Iso}(X,Y), \mathbf{Iso}(X)$	Isometries from $\ell^2(X)$ to $\ell^2(Y)$ (on $\ell^2(X)$)	
$\mathbf{Iso}[V,W], \mathbf{Iso}(V)$	Isometries from $\ell^2[V]$ to $\ell^2[W]$ (on $\ell^2[V]$)	
$\mathbf{x} \leftarrow \mathbf{measure}\ \mathbf{q}_1 \dots \mathbf{q}_n\ \mathbf{with}\ e$	Statement: Measure quantum variables $\mathbf{q}_1 \dots \mathbf{q}_n$ with measurement e	
$\mathbb{B}$	Booleans. $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$	
$\ell^2(B)$	Hilbert space with basis indexed by B	
$\ell^2[V]$	$\ell^2(\mathsf{Type}_V^{\mathsf{set}})$ – Hilbert space with basis $\mathsf{Type}_V^{\mathsf{set}}$	
$\mathbf{U}(X,Y), \mathbf{U}(X)$	Unitaries from $\ell^2(X)$ to $\ell^2(Y)$ (on $\ell^2(X)$)	
$\mathbf{U}[V,W], \mathbf{U}(V)$	Unitaries from $\ell^2[V]$ to $\ell^2[W]$ (on $\ell^2[V]$)	
$fv(e)$	Free variables in an expression e (or program)	
$\mathsf{Type}_e^{\mathsf{exp}}$	Type of an expression e	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{uniform}}$	qRHL judgment, uniform definition	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{nonsep}}$	qRHL judgment, non-separable definition	
A^*	Adjoint of the operator A	
id	Identity	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}$	Quantum relational Hoare judgment	
$\|b\rangle, \|b\rangle_V$	Basis vector in Hilbert space $\ell^2[V]$	
$\mathbf{B}^{\leq 1}(X,Y)$	Bounded linear operators with operator norm ≤ 1	
$\mathbf{B}(X,Y)$	Bounded linear operators from $\ell^2(X)$ to $\ell^2(Y)$	
id_V	Identity on $\ell^2[V]$ or on $\mathbf{B}[V]$	
$\mathbf{B}[V,W]$	Bounded linear operators from $\ell^2[V]$ to $\ell^2(W)$	
$R_1 \circ R_2$	Composition of relations	
$a \circ_e b$	Composition of expressions a, b as relations	
$A \div \psi$	Part of A containing ψ	
$m_1 m_2$	Union (concatenation) of memories m_1, m_2	
`INF x:Z. e`	Intersection of family of subspaces (Isabelle/HOL syntax)	8
$A \sqcup B$	Sum of subspaces (Isabelle/HOL syntax)	8
`top`	Full subspace (Isabelle/HOL syntax)	8
`bot`	Zero subspace (Isabelle/HOL syntax)	8
$\mathbf{supp}\, M$	Support of an operator M	
$\mathsf{marginal}_i(\mu)$	i-th marginal distribution of μ (for $\mu \in \mathbf{D}^{\leq 1}(X \times Y)$, $i = 1, 2$)	
$x \leftarrow e$	Program: assigns expression e to x	
$\operatorname{supp} \mu$	Support of distribution μ	
$\mathbf{Meas}(D,E)$	Projective measurements on $\ell^2(E)$ with outcomes in D	
$U_{vars,Q}$	Canonical isomorphism between $\ell^2(\mathsf{Type}_Q^{\mathsf{list}})$ and $\ell^2[\mathsf{Type}_Q^{\mathsf{set}}]$ for a list Q	
$S^\perp$	Orthogonal complement of subspace S	
$\mathrm{idx}_1\, \mathbf{c}, \mathrm{idx}_1\, e$	Add index 1 to every variables in $\mathbf{c}$ or e	
V^{cl}	Classical variables in V	
$\mathsf{Type}_V^{\mathsf{list}}$	Type of a list V of variables	
$\mathrm{Time}(A)$	Worst-case runtime of A	23
$A \sqcap B$	Intersection of subspaces (Isabelle/HOL syntax)	8
$P \div \psi \gg Q$	Isabelle/HOL syntax for `space_div` (related to $P \div \psi$)	14
$\lfloor x \rfloor$	x rounded down to the next integer	
$\mathtt{Qeq}[\mathbf{q}_1, \dots, \mathbf{q}_n = \mathbf{q}'_1, \dots, \mathbf{q}'_m]$	Isabelle/HOL syntax for quantum equality $\mathbf{q}_1, \dots, \mathbf{q}_n \equiv_{\mathsf{quant}} \mathbf{q}'_1, \dots, \mathbf{q}'_m$	14
$[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$	Typed tuple of quantum variables (Isabelle/HOL syntax)	12
$\mathrm{Adv}^{A_1 A_2}_{\mathrm{IND}}(\eta)$	Advantage of adversary A_1, A_2 in IND-OT-CPA game with security parameter η	25

$\mathrm{Adv}^{A}_{\mathrm{PRG}}(\eta)$	Advantage of adversary A in PRG-OT-CPA game with security parameter η	23
$x \in_{\$} M$	x uniformly sampled from M	22
$\mathrm{Adv}^{A_1 A_2}_{\mathrm{ROR}}(\eta)$	Advantage of adversary A_1, A_2 in ROR-OT-CPA game with security parameter η	22
$\mathbf{T}^{+}[V]$	Positive trace class operators on $\ell^2[V]$	
$\mathbf{T}[V]$	Trace class operators on $\ell^2[V]$	
$\mathbb{R}$	Real numbers	
$\mathbb{C}$	Complex numbers	
$\mathbf{D}^{\leq 1}[V]$	Sub-probability distributions over variables V	
$\mathbf{D}^{\leq 1}(X)$	Sub-probability distributions over X	
$A \otimes B$	Tensor product of vectors/operators A and B	
$\mathbf{T}^{+}(X)$	Positive trace class operators on $\ell^2(X)$	
$\mathbf{T}(X)$	Trace class operators on $\ell^2(X)$	
$\mathbb{R}_{\geq 0}$	Non-negative real numbers	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{class}}$	pRHL judgement (classical)	
$\mathrm{tr}^{[V]}_{W}(\rho)$	Partial trace, keeping variables V, dropping variables W	
$\mathrm{im}\, A$	Image of A	
$\mathrm{dom}\, f$	Domain of f	
$\|x\|$	ℓ_2-norm of vector x, or operator-norm	
$\mathrm{tr}\, M$	Trace of matrix/operator M	
$\mathfrak{Cla}[e]$	Classical predicate meaning $e = \texttt{true}$	
$\mathrm{span}\, A$	Span, smallest subspace containing A	
$X_1 \equiv_{\mathsf{quant}} X_2$	Equality of quantum variables X_1 and X_2	
$\mathsf{proj}(x)$	Projector onto x, i.e., xx^*	
V^{qu}	Quantum variables in V	
$\mathsf{Type}^{\mathsf{set}}_X$	Type of a set V of variables	
$Q_1 \equiv\mathtt{q}\ Q_2,\ Q_1\ \mathtt{==q}\ Q_2$	Isabelle/HOL syntax for quantum equality $Q_1 \equiv_{\mathsf{quant}} Q_2$	14
$A\mathtt{»}Q$	Overloaded Isabelle/HOL constant for `liftOp`, `liftSpace`	12, 12
$\otimes$	Tensor product (Isabelle/HOL constant)	11, 11, 14
2^M	Powerset of M	
$x\ \mathtt{<\$}\ e\mathtt{;}$	Sampling (tool program syntax)	5
$x\ \mathtt{<-}\ e\mathtt{;}$	Assignment (tool program syntax)	4
$\mathtt{Pr}[v : P(\rho)]$	Isabelle/HOL constant for probability of $v = 1$ after running P	10
$\mathbf{q}_1, \ldots, \mathbf{q}_n\ \mathtt{<q}\ e\mathtt{;}$	Quantum initialization (tool program syntax)	5
$[\![\mathbf{q}_1, \ldots, \mathbf{q}_n]\!]$	Typed tuple of quantum variables (Isabelle/HOL syntax)	
$A \cdot B$	Overloaded Isabelle/HOL constant for `timesOp`, `applyOp`, `applyOpSpace`	10, 10, 10
$\mathbf{d}$	A program	
$\mathbf{c}$	A program	
$[\![\mathbf{c}]\!]_{\mathbf{class}}$	Classical denotation of a program $\mathbf{c}$	
$[\![\mathbf{c}]\!]$	Denotation of a program $\mathbf{c}$	
if e **then** c_1 **else** c_2	Statement: If (conditional)	
skip	Program that does nothing	
$\mathbf{x} \xleftarrow{\$} e$	Statement: Sample $\mathbf{x}$ according to distribution e	
while e **do** c	Statement: While loop	
apply $\mathbf{q}_1 \ldots \mathbf{q}_n$ **to** U	Statement: Apply unitary/isometry U to quantum registers $\mathbf{q}_1 \ldots \mathbf{q}_n$	
$\mathbf{q}_1 \ldots \mathbf{q}_n \xleftarrow{\mathbf{q}} e$	Statement: Initialize $\mathbf{q}_1, \ldots, \mathbf{q}_n$ with quantum state e	
$A\mathtt{»}Q$	Lifts operator or subspace to variables Q	

$\mathcal{E}$	A superoperator
$\mathbf{T}_{cq}^{+}[V]$	Positive trace class cq-operators on $\ell^2[V]$
$\mathsf{lift}(\mu)$	Transforms a distribution μ into a density operator
`true`	Truth value "true"
`false`	Truth value "false"
$\downarrow_e(\rho)$	Restrict state/distribution ρ to the case $e =$ `true` holds
$\mathbf{c}; \mathbf{d}$	Sequential composition of programs
$\mathbf{T}_{cq}[V]$	Trace class cq-operators on $\ell^2[V]$
δ_x	Point distribution: returns x with probability 1
$\|x\|$	Absolute value of x / cardinality of set x
Type_v	Type of variable v
$[\![e]\!]_m$	Denotation of a classical expression e, evaluated on classical memory m
$\mathbf{y}$	A classical program variable
$\mathbf{x}$	A classical program variable
$f\|_M$	Restriction of function f to domain M
$e\{f/\mathbf{x}\}$	Substitute f for variable $\mathbf{x}$ in e
$e\sigma, \mathbf{c}\sigma$	Apply variable renaming σ to expression e
$\mathcal{E}_{rename,\sigma}$	cq-superoperator: Renames variables according to bijection σ
$U_{rename,\sigma}$	Unitary: Renames variables according to bijection σ
$\Pr[e : \mathbf{c}(\rho)]$	Probability that e holds after running $\mathbf{c}$ on initial state ρ
$f(x := y)$	Function update, i.e., $(f(x := y))(x) = y$
$\mathbf{q}$	A quantum program variable

Index