qRHL tool – Manual

Dominique Unruh
University of Tartu

Contents

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 [12].

The source code is published on GitHub [10].

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

1 Architecture

The tool consists of three main components: a ProofGeneral [8] frontend, the core tool written in Scala, and a Isabelle/HOL [6] 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 [5]).[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 `{ℭ𝔩𝔞[x1=x2]} x <- x+1; ~ skip; {ℭ𝔩𝔞[x1≠x2]}`, the predicates `ℭ𝔩𝔞[x1=x2]` and `ℭ𝔩𝔞[x1≠x2]` are also parsed by Isabelle/HOL. In order to support the different constructions used in predicates (see Section 4 in [12], e.g.,

[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.

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.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.

$\mathfrak{Cla}[\dots]$ 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`.

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. Several theories can be specified in one `isabelle` command (comma separated). Repeated *identical* `isabelle` commands are allowed. See Section 6 for more information on accompanying theories.

[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`.

Including files. The `include` command allows us to include a another qrhl file. "`include "filename".` includes the file `filename`. The effect of including a file is the same as directly copying its content into the current file, with two differences:

- A command to include a file that has already been included will be ignored. This means that several files can include the same file without duplicating declarations, allowing for a dag dependency structure.
- In interactive mode (i.e., in ProofGeneral), the content of an included file is executed in "cheat mode". That is, the proofs in those files are assumed to be correct and not checked. This speeds up development. (To check a file `file.qrhl` and all recursively included files, use the command line `bin/qrhl file.qrhl`.)

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 [12]. 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.

Types used with `classical var` and `quantum var` must satisfy an important condition. Namely, they must be in the type class[3] `universe`.[4] For most predeclared types, this will already be the case, but if not, follow the instructions in Section 6.1.

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 free v1,v2,v3,...,vn.
```

That is, after this declaration, `name` is assumed to refer to some program containing (at most) the free 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.

In some cases, an adversary may invoke other programs (e.g., an encryption oracle). In that case, we declare an adversary with "holes" using:

[3] A type class represents a property of a type, for example, the type class `finite` applies only to types with finite domain.

[4] The type class `universe` guarantees that the type is small enough (its cardinality is at most $\beth_i$ for some $i \in \mathbb{N}$). Without this restriction, it would be possible, e.g., to have a program variable of type P `set`, where P is the type of all programs. That would mean that programs can contain arbitrary elements of P `set`. Hence the powerset of P can be embedded in P which is impossible. Restricting program variables to small types makes it possible to define P (and related types). This is not a restriction in practice since all types built from basic types using powersets, functions, and inductive datatypes are small in this sense.

```
adversary name free v1,v2,v3,...,vn calls ?,?,...,?.
```

This means the program `name` contains variables `v1,v2,v3,...,vn`, as well as n "holes" (one for each question mark).[5] We write `name(p1,...,pn)` to refer to `name` with `p1`,...,`pn` inserted instead of the holes. (For example, `name(enc,dec)` would run the adversary `name` and allow it to invoke the programs `enc` and `dec`.) Note that the variables `v1,v2,v3,...,vn` do not have to include variables contained in the programs `p1,...,pn`.

In addition to specifying the free variables of `name`, the `adversary` command allows us to specify various other variable sets. The full syntax of the command is:

```
adversary A free F
          readonly R
          overwritten O
          inner I
          covered C
          calls ?,?,...,?.
```

All variable specifications except `free` F are optional but they have to occur in this order. F are (an upper bound on) the free variables of `name`. A. R are a lower bound on the readonly variables.[6] O is a lower bound on the overwritten variables, that is, variables in O are guaranteed to be overwritten before they are read by A (or any of the oracles invoked by A). I is an upper bound on the inner variables of A, that is, all local variables that have an oracle call in their scope. C are a lower bound for the covered variables of A, i.e., those variables that are local over *every* hole of A. Precise definitions of these variable sets are given in [11]. The default value for all these variable sets are the empty set.

These variable sets are necessary to avoid certain subtleties involving oracle calls. For example, an oracle O may access a global variable `x`, but the adversary may invoke O in a context where `x` is declared as a local variable. This will hide the global state of `x` from O. Thus we need to know for which variables this happens. This is precisely described by the inner variables I.

Note that besides declaring the different variable sets, the user does not have to care about them. When defining programs explicitly (using the `program` command), all variable sets will be automatically derived. Some tactics such as the `frame` and the `equal` tactic heavily rely on the various variable sets to decide whether they can be applied.

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 free 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:

[5] Formally, it declares `name` to refer to a multi-hole context with n holes in the sense of footnote 17 in [12].
[6] That is, $F \setminus R$ is an upper bound for the written variables of A are at most.

```
# 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 name: {pre} code1 ~ code2 {post}.
```

Here `name` is the name under which the proven fact will be stored. And `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 $\{\texttt{pre}\}\texttt{code1} \sim \texttt{code2}\{\texttt{post}\}$.

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

```
lemma name: formula.
```

Here `formula` is an arbitrary formula that Isabelle/HOL understands (ambient logic). 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 [12]). 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 $A_1, \dots, A_n$ 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.[7] Each statement is one of the following:

[7]This deviates slightly from the syntax of programs described in Section 3.2 in [12]. 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.

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`.
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 $q_1, \dots, q_n \xleftarrow{\mathtt{q}}$ *expr*. $q_1, \dots, q_n$ must be declared as quantum variables with some Isabelle/HOL types $T_1, \dots, T_n$. All q_i must be distinct variables. *expr* must be an Isabelle/HOL expression of type $(T_1 \times \cdots \times T_n)$ `ell2`, 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 [12]) 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).[8] **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 $x \leftarrow$ **measure** $q_1, \dots, q_n$ **with** *measurement* x must be declared as a classical variable of some Isabelle/HOL type T_x. $q_1, \dots, q_n$ must be declared as quantum variables of some Isabelle/HOL types $T_1, \dots, T_n$. 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.

[8] Formally, changing the type of programs is justified as follows: A program q_1`,...,`q_n `<q` *expr*`;` is interpreted as $q_1, \dots, q_n \xleftarrow{\mathtt{q}} mkUnit(e)$ 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 [12]. See footnote 21 for how this affects the rules implemented by the tactics.

`on` q_1`,`...`,`q_n `apply` *expr*`;`	The unitary quantum operation **apply** *expr* **to** $q_1, \dots, q_n$. $q_1, \dots, q_n$ must be declared as quantum variables of some Isabelle/HOL types $T_1, \dots, T_n$. All q_i must be distinct variables. *expr* must be an Isabelle/HOL expression of type $(T_1 \times \dots \times T_n, T_1 \times \dots \times T_n)$ `l2bounded`, 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 [12]) 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).[9] **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.

[9] Formally, changing the type of programs is justified as follows: A program `on` $q_1, \dots, q_n$ `apply` e`;` is interpreted as **apply** $mkIso(e)$ **to** $q_1, \dots, q_n$ 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 [12]. See footnote 20 for how this affects the rules implemented by the tactics.

`{ local` X`; P }`	Declares local variables X in the program P. X is a comma separated list of quantum and classical variables. Those variables must have been declared using the `classical`/`quantum var` command. The variables X will then be local in the program P. P may be a single or several statements. Semantically, `{ local` X`; P }` first stores the variables X on a stack and initializes them with a default value, then runs P, and then restores the original value of X. **Example:** "`{ local z; z <- x; x <- y; y <- z }`" swaps `x` and `y` without any side effect on `z`.
`call` *prog*`;`	The program *prog* itself. *prog* must be the name of a program (declared with `program` or `adversary`). If the `adversary`-command declared a program "with n holes" (using `adversary ... calls ?,...,?`), then *prog* is an expression of the form `name(arg1,...,argn)` where each `arg1,...,argn` is again the name of a program (or an expression of the form `name(arg1,...,argn)`). Logically, `call` *prog*`;` is simply an abbreviation for the code of *prog* (possibly after substituting `arg1,...,argn` for its holes). 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 [12] 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 ...`). "`call A(enc,dec);`" invokes the adversary `A` that can call programs `enc` and `dec` (assuming `A` was declared using `adversary A vars ... calls ?,?`).

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 [7].

For experiments, it can be useful to directly invoke Isabelle/HOL (using the `./run-isabelle.sh` script) and edit a theory that imports `QRHL.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) `l2bounded`, α `ell2`, 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 clinear_space` (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 clinear_space` is the type of closed subspaces of $\ell^2[V_1V_2]$ where 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.

When defining your own types in an accompanying theory, please consult Section 6.1.

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 `ell2`	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 `clinear_space`	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 clinear_space`, 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) `cblinfun`	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). This type will almost always be used as $(\alpha'$ `ell2`$, \beta'$ `ell2`$)$ `cblinfun`). (I.e., elements of $\mathbf{B}[\alpha, \beta]$.) This can be abbreviated as (α', β') `l2bounded`.
(α, β) `l2bounded`	Abbreviation for $(\alpha$ `ell2`$, \beta$ `ell2`$)$ `cblinfun`).
(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`.[10] 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 $\mathtt{q}_1, \dots, \mathtt{q}_n$ are variables of types α_1 `variable`,...,α_n `variable`, then their tuple (constructed with the syntax $[\![\mathtt{q}_1, \dots, \mathtt{q}_n]\!]$) 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 $\mathbf{T}^+_{cq}[V_1 V_2]$ 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**

[10] `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.

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` $\mu = 1$.
`prob` μ x :: `real` (for μ :: α `distr` and x :: α)	The probability $\mu(x)$ of x according to distribution μ.
`point_distr` x :: α `distr` (for x :: α)	Probability distribution that samples x with proability 1. That is, $\mu(y) = 1$ if $y = x$ and $\mu(y) = 0$ otherwise for $\mu :=$ `point_distr` x.
`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.
`bind_distr` μ f :: β `distr` (for μ :: α `distr` and $f :: \alpha \Rightarrow \beta$ `distr`)	The distribution of y if x is sampled according to μ and y according to $f(x)$. (Monadic bind.) That is, $\nu(y) = \sum_x \mu(x)f(x)(y)$ for $\nu :=$ `bind_distr` f μ. In particular, `map_distr` f μ = `bind_distr` μ ($\lambda x.$ `point_distr` $(f\ x)$).
`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`$[e : P(\rho)]$ :: `real` (for e :: `bool` and P :: `program` and ρ :: `program_state`)	The probability $\Pr[e : P(\rho)]$ that $e =$ `true` after execution of the program P with initial state ρ. Here e must be an expression of type `bool` (and e may contain ambient and program variables without indices). 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`$[e : P(\rho)]$. It should not be used directly.

Operators

A^* `adjoint` A :: (β, α) `cblinfun` (for A :: (α, β) `cblinfun`)	The adjoint A^* of A.
$A \cdot B$ `timesOp` A B :: (α, γ) `cblinfun` (for A :: (β, γ) `cblinfun` and B :: (α, β) `cblinfun`)	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`
$A \cdot \psi$ `cblinfun_apply` A ψ :: β (for A :: (α, β) `cblinfun` and ψ :: α)	The result $A\psi$ of applying the operator A to the ell2 ψ. The syntax $A \cdot \psi$ is overloaded. If Isabelle/HOL has trouble recognizing which meaning of $\cdot$ is intended, use `cblinfun_apply`, or provide additional type information for the A and ψ. **How to input:** `\cdot`

$A \cdot S$ `applyOpSpace` A S :: β `clinear_space` (for $A :: (\alpha, \beta)$ `cblinfun` and $S :: \alpha$ `clinear_space`)	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) `cblinfun`	The identity operator id on $\ell^2(\alpha)$.
`addState` ψ :: $(\beta, \beta \times \alpha)$ `l2bounded` (for $\psi :: \alpha$ `ell2`)	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 [12].)
`unitary` A :: `bool` (for $A :: (\alpha, \beta)$ `cblinfun`)	True iff A is unitary.
`isometry` A :: `bool` (for $A :: (\alpha, \beta)$ `cblinfun`)	True iff A is an isometry.
`isProjector` A :: `bool` (for $A :: (\alpha, \alpha)$ `cblinfun`)	True iff A is a projector.
`Proj` S :: (α, α) `cblinfun` (for $S :: \alpha$ `clinear_space`)	The projector onto subspace S.
`proj_classical_set` S :: (α, α) `l2bounded` (for $S :: \alpha$ `set`)	The projector onto the span of $\lvert s\rangle$ with $s \in S$. (Equivalently $\sum_{s \in S} \lvert s\rangle\langle s\rvert$.)
`hadamard,pauliX,pauliY,pauliZ` :: (`bit`, `bit`) `l2bounded`	Hadamard, or Pauli X, Y, Z operators, respectively.
`CNOT` :: (`bit` $\times$ `bit`, `bit` $\times$ `bit`) `l2bounded`	Controlled-not on two qubits (first qubit is the control)
$A \otimes B$ `tensor` A B `tensorOp` A B :: (α, β) `l2bounded` (for $A :: \alpha$ `l2bounded` and $B :: \beta$ `cblinfun`)	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 [12]. 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, \ox`
`comm_op` :: $(\alpha \times \beta, \beta \times \alpha)$ `l2bounded`	The canonical isomorphism between $\ell^2(X \times Y)$ and $\ell^2(Y \times X)$. That is, the operator mapping $\lvert x, y\rangle$ to $\lvert y, x\rangle$.
`assoc_op` :: $(\alpha \times (\beta \times \gamma), (\alpha \times \beta) \times \gamma)$ `l2bounded`	The canonical isomorphism between $\ell^2(X \times (Y \times Z))$ and $\ell^2((X \times Y) \times Z)$. That is, the operator mapping $\lvert x, (y, z)\rangle$ to $\lvert (x, y), z\rangle$. 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.

`Uoracle` f :: $(\alpha \times \beta, \alpha \times \beta)$ `l2bounded` (for $f :: \alpha \Rightarrow \beta$)	Classical function f represented as a unitary. More precisely, `Uoracle` $f : \lvert(x, y)\rangle \mapsto \lvert(x, y + f(x))\rangle$. (This is a useful construct when modeling, e.g., function that can be queried in superposition by a quantum algorithm.) The type β must have sort `group_add`. This guarantees that $y+f(x)$ is well-defined and has suitable properties (in particular, this makes `Uoracle` f unitary). If β has even sort `xor_group` (Abelian group with $x+x = 0$), then additional laws for `Uoracle` will be available. Examples of types that have these sorts are `bit`, `int`, $(\dots,\dots)$ `nlist`. (The latter is defined in CryptHOL [2], but you additionally need to import the theory `QRHL.CryptHOL_Missing`.) When axiomatizing a type `T`, use the `declare_variable_type` command in Isabelle to ensure that it has the relevant sorts. (See Section 6.1.)

States

$\lvert x\rangle$ `ket` x :: α `ell2` (for $x :: \alpha$)	The basis state $\lvert x\rangle$ of $\ell^2(\alpha)$. **How to input:** `\ket`
`EPR` :: (`bit` $\times$ `bit`) `ell2`	The state $\frac{1}{\sqrt{2}}\lvert 00\rangle + \frac{1}{\sqrt{2}}\lvert 11\rangle$.
$\psi \otimes \phi$ `tensor` ψ ϕ `tensorVec` ψ ϕ :: (α, β) `ell2` (for $\psi :: \alpha$ `ell2` and $\phi :: \beta$ `ell2`)	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 [12]. 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, \ox`

Quantum variables

$[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$ $[\lvert\mathbf{q}_1, \dots, \mathbf{q}_n\rvert]$ :: $(\alpha_1 \times \cdots \times \alpha_n)$ `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`) `l2bounded` (for $A :: (\alpha, \alpha)$ `l2bounded` and $Q :: \alpha$ `variables`)	The operator A»$Q := U_{vars,Q} A U^*_{vars,Q} \otimes id_{V_1^{\mathsf{qu}} V_2^{\mathsf{qu}} \setminus Q}$. (See Definition 19 in [12].) 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 `clinear_space` and Q :: α `variables`)	The subspace S»$Q := U_{vars,Q} S \otimes \ell^2[V_1V_2 \setminus Q]$. (See Definition 19 in [12].) 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` $[\![\dots]\!]$ 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` $[\![\dots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.
`colocal` A Q `colocal_op_qvars` A Q :: `bool` (for A :: (`mem2`, `mem2`) `l2bounded` 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` $[\![\dots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.
`colocal` A P `colocal_op_pred` A P :: `bool` (for A :: (`mem2`, `mem2`) `l2bounded` 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` $[\![\dots]\!]$ to the Isabelle simplifier, see the explanations for `declared_qvars`.

declared_qvars $[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$ declared_qvars $[\|\mathbf{q}_1, \dots, \mathbf{q}_n\|]$ :: 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 $\mathbf{q}_1 = s_1 \wedge \dots \wedge$ 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 $[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$ 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 $[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$ (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 $[\![\dots]\!]$ are solved automatically by the simp tactic.

Subspaces & predicates

span M :: α clinear_space (for M :: α ell2 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:** \Cla
quantum_equality_full $A_1\, Q_1\, A_2\, Q_2$ :: predicate (for A_1 :: (α, γ) l2bounded and Q_1 :: α variables and A_2 :: (β, γ) l2bounded and Q_2 :: β variables)	The quantum equality predicate $A_1 Q_1 \equiv_{\mathsf{quant}} A_2 Q_2$. (Definition 27 in [12])
$Q_1 \equiv\mathfrak{q}\ Q_2$ 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 28 in [12]) 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 $\alpha_1 \times \dots \times \alpha_n = \alpha'_1 \times \dots \times \alpha'_m$. **How to input:** \qeq

$P \div \psi » Q$ `space_div` P ψ Q :: `predicate` (for P :: `predicate` and ψ :: α `ell2` 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 `clinear_space` (for S :: α `clinear_space`)	Orthogonal complement $S^{\perp}$ of S.
$S \otimes T$ `tensor` S T `tensorSpace` S T :: (α, β) `clinear_space` (for S :: α `clinear_space` and T :: β `clinear_space`)	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 [12]. 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, \ox`

Measurements

`binary_measurement` P :: (`bit`, α) `measurement` (for P :: (α, α) `l2bounded`)	Constructs a binary measurement from the project P. (I.e., outcome 1 corresponds to P and outcome 0 to $1 - P$.)
`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) `l2bounded` (for M :: (α, β) `measurement` and x :: α)	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 [12]. 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

When invoked as "`byqrhl qvars` $\mathbf{q}^{(1)},\dots,\mathbf{q}^{(m)}$.", transforms a goal of the form $\mathtt{Pr}[e : P(\rho)] = \mathtt{Pr}[e' : P'(\rho)]$ into a qRHL subgoal. (Also works for $\leq$ or $\geq$ instead of $=$.)

Here e, e' must be expressions of type `bool` (that may contain classical and ambient variables), 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:

$$\frac{\big\{\mathfrak{Cla}[\mathbf{y}_1^{(1)} = \mathbf{y}_2^{(1)} \wedge \dots \wedge \mathbf{y}_1^{(n)} = \mathbf{y}_2^{(n)}] \sqcap [\![\mathbf{q}_1^{(1)},\dots,\mathbf{q}_1^{(m)}]\!] \equiv\mathfrak{q}\ [\![\mathbf{q}_2^{(1)},\dots,\mathbf{q}_2^{(m)}]\!]\big\}\mathtt{call}\ P \sim \mathtt{call}\ P'\big\{\mathfrak{Cla}[e_1 \leftrightarrow e_2']\big\}}{\mathtt{Pr}[e : P(\rho)] = \mathtt{Pr}[e' : P'(\rho)]}$$

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

Here $\mathbf{y}^{(1)},\dots,\mathbf{y}^{(n)}$ are the free classical variables of P, P', e, e'. And $\mathbf{q}^{(1)},\dots,\mathbf{q}^{(m)}$ are required to be a superset of the quantum variables in $\big(\mathit{fv}(P) \setminus \mathit{overwr}(P)\big) \cup \big(\mathit{fv}(P') \setminus \mathit{overwr}(P')\big)$. ($\mathit{fv}(P)$ are the free variables, and $\mathit{overwr}(P)$ the overwritten variables of P.)[11]

If the tactic is invoked simply as `byqrhl`, then $\mathbf{q}^{(1)},\dots,\mathbf{q}^{(m)}$ will simply be the quantum variables in $\big(\mathit{fv}(P) \setminus \mathit{overwr}(P)\big) \cup \big(\mathit{fv}(P') \setminus \mathit{overwr}(P')\big)$, i.e., the minimum allowed set of quantum variables.

If the conclusion contains $\leq$ or $\geq$ instead of $=$, then $\leftrightarrow$ is replaced by $\rightarrow$ or $\leftarrow$, respectively. If $m = 0$, then $[\![\mathbf{q}_1^{(1)},\dots,\mathbf{q}_1^{(m)}]\!] \equiv\mathfrak{q}\ [\![\mathbf{q}_n^{(1)},\dots,\mathbf{q}_n^{(m)}]\!]$ is replaced by `top`.

The rule is a special case of rule QrhlElimEqNew in [11].

As a special case, the lhs or rhs can also be "1" instead of an expression $\mathtt{Pr}[e : P(\rho)]$. This is then interpreted as if the lhs/rhs was $\mathtt{Pr}[\mathtt{True} : \mathtt{skip}(\rho)]$.

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 casesplit

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

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

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 $A_1 \rightarrow \dots \rightarrow A_m \rightarrow B$, A_n is considered to be the n-th assumption.

[11] Those sets are defined in [11]. The command "`print` P" shows those variables.

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

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`):

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

Both rules are special cases of rule Conseq.

An alternative invocation is "`conseq qrhl:` *lemma*". In this case, *lemma* has must be the name of an already proven theorem (using the `qrhl` command) stating $\{A'\}\mathbf{c} \sim \mathbf{d}\{B'\}$. Then `conseq qrhl:` *lemma* applies the rule:

$$\frac{A \le A' \qquad B' \le B}{\{A\}\mathbf{c} \sim \mathbf{d}\{B\}}$$

That is, this form can be used when the current qRHL judgment has already been proven, except with slightly different pre-/postconditions. (But the programs need to be identical.)

This is still a special case of rule Conseq.

In many cases, already proven qRHL judgments *lemma* are of the form $\{A \sqcap L \equiv_{\mathsf{quant}} R\}\mathbf{c} \sim \mathbf{d}\{B \sqcap L' \equiv_{\mathsf{quant}} R'\}$ where the variables in the quantum equality are not exactly the ones needed in the present subgoal. In this case, we can use the tactic "`conseq qrhl (`L_{old}`->`L_{new}`;` R_{old}`->`R_{new}`):` *lemma*". In this form, the tactic will first attempt to rewrite the quantum equality *lemma*: In both L, L', L_{old} is replaced by L_{new}, and in R, R', R_{old} is replaced by R_{new}. Then the tactic behaves like "`conseq qrhl:` *lemma*" above except that the rewritten lemma is used.

For the rewriting to be possible, the following conditions need to be satisfied:

- L_{old} is a suffix of both L, L'. R_{old} is a suffix of both R, R'. (Checked by the tactic.)
- $(L_{old} \cup L_{new}) \cap fv(\mathbf{c}) = \varnothing$. (Checked by the tactic.)
- $(R_{old} \cup R_{new}) \cap fv(\mathbf{d}) = \varnothing$. (Checked by the tactic.)
-
  - $|\mathsf{Type}^{\mathsf{set}}_{L_{new}}| = \infty \lor |\mathsf{Type}^{\mathsf{set}}_{L_{new}}| \ge |\mathsf{Type}^{\mathsf{set}}_{L_{old}}|$.
  - $|\mathsf{Type}^{\mathsf{set}}_{R_{new}}| = \infty \lor |\mathsf{Type}^{\mathsf{set}}_{R_{new}}| \ge |\mathsf{Type}^{\mathsf{set}}_{R_{old}}|$.
  - L_{old}, L_{new} (indexed with 1) and R_{old}, R_{new} (indexed with 1) are disjoint from the free variables of A, B.

  (These three conditions are returned as a single subgoal, usually easy to solve using `simp`.)

The rewriting is justified by rule EqVarChange in [11].

If any of $L_{old}, L_{new}, R_{old}, R_{new}$ should be the empty list, then the notation "`.`" can be used. (E.g., `x,y->.` means variables x, y are simply removed.)

When invoking "`conseq qrhl (`V_{old}`->`V_{new}`):` *lemma*", this is short for "`conseq qrhl (`V_{old}`->`V_{new}`;` V_{old}`->`V_{new}`):` *lemma*". (Same replacement on left/right side.)

To be able to use this tactic, it is a good idea to set aside a variable `aux` of some infinite type[12] that never occurs in any programs, and then to always prove qRHL judgments of the form $\{A \sqcap L \equiv_{\mathsf{quant}} R\}\mathbf{c} \sim \mathbf{d}\{B \sqcap L' \equiv_{\mathsf{quant}} R'\}$ where L, R, L', R' all end in `aux`. (Intuitively, this means the judgment in question also preserves equality of an uninvolved variable `aux`.) Then `aux` can be replaced by other quantum variables as needed when the qRHL judgement is used in a subproof.

Tactic equal

Converts a subgoal of the form $\{A\}\mathbf{c}_0; \mathbf{c} \sim \mathbf{d}_0; \mathbf{d}\{B\}$ where $\mathbf{c}, \mathbf{d}$ satisfy $\mathbf{c} = \mathbf{d}$ (up to few differences) into a subgoal $\{A\}\mathbf{c}_0 \sim \mathbf{d}_0\{D\}$ with suitably updated postcondition D. In addition, a subgoal about free variables, as well as subgoals corresponding to the differences between s, s' (if any) are produced.

[12] Such a variable `aux` is predeclared by the tool.

The simplest form is to invoke the tactic as `equal`, this removes the last statement on both sides, assuming it is the same statement.

In general, the tactic is invoked as: "`equal` n `exclude` P `in` V_{in} `mid` V_{mid} `out` V_{out}`.`"

Here n denotes how many statements should be included in the suffix $\mathbf{c}/bd$. n can be a natural number (meaning the last n lines should be removed), the keyword `all` (meaning the whole left/right program should be removed), or omitted (meaning one line should be removed).

P is a comma-separated list of program names. When the tactic identifies where $\mathbf{c}, \mathbf{d}$ differ (see below), all invocations of the programs P are included in the list of differences (even if they are the same invocation on both sides). This can be useful if the programs in P contain variables that would get included in the invariants generated by `equal` in an undesired way. (Instead, we get extra subgoals for those programs that we need to prove manually.) "`exclude` P" can be omitted.

The equal tactic works by maintaining an invariant throughout $\mathbf{c}, \mathbf{d}$ that all relevant variables are equal on the left/right side. Which variables are included in those invariants can be finetuned using the comma-separated variable lists V_{in}, V_{mid}, V_{out}. V_{in} specifies which variables should be equal before $\mathbf{c}, \mathbf{d}$. (I.e., V_{in} occurs in the updated postcondition D.) V_{mid} specifies which variables should be equal during the execution of $\mathbf{c}, \mathbf{d}$ (this will affect the invariants in subgoals corresponding to the differences between $\mathbf{c}, \mathbf{d}$). V_{out} specifies which variables should be equal after execution of $\mathbf{c}, \mathbf{d}$ (this affects how the original postcondition B is treated, in particular, if B contains a quantum equality, then V_{out} should contain exactly the quantum variables in that quantum equality).

The sets V_{in}, V_{mid}, V_{out} must satisfy a number of conditions. If those conditions are not satisfied, the tactic tries to add as few variables as possible to these sets so that all conditions are met. (The tactic also outputs a log explaining which variables are added to make which condition true.) Each of the specifications `in` V_{in}, `mid` V_{mid}, and `out` V_{out} can be omitted. This means the tactic includes as few variables as possible in the corresponding variable list.

In detail:

The tactic works by instantiating and applying the following rule from [11]:

$$\textsc{Adversary}\quad \frac{V_{in}, V_{mid}, V_{out} \text{ satisfy numerous conditions (see [11])} \qquad \forall i. \big\{R \cap {\equiv}V_{mid}\big\} s_i \sim s'_i \big\{R \cap {\equiv}V_{mid}\big\}}{\big\{R \cap {\equiv}V_{in}\big\} C[s_1, \dots, s_n] \sim C[s'_1, \dots, s'_n] \big\{R \cap {\equiv}V_{out}\big\}}$$

(Here $\equiv V$ denotes $\mathfrak{Cla}[\mathbf{x}_1^{(1)} = \mathbf{x}_2^{(1)} \land \dots \land \mathbf{x}_1^{(n)} = \mathbf{x}_2^{(n)}] \cap Q_1 \equiv_{\mathsf{quant}} Q_2$ where $\mathbf{x}^{(i)}$ are the classical variables in V and Q are the quantum variables in V, and Q_1, Q_2 are Q indexed with $1/2$, respectively.)

By comparing $\mathbf{c}, \mathbf{d}$, a context C with multiple holes is obtained such that $\mathbf{c} = C[s_1, \dots, s_n]$ and $\mathbf{d} = C[s'_1, \dots, s'_n]$.[13] It is furthermore guaranteed that no program in P occurs in C. (In particular, if $\mathbf{c} = \mathbf{d}$ and $P = \varnothing$, then simply $C = \mathbf{c} = \mathbf{d}$.) We call the s_i, s'_i pairs "mismatches".

Next, the tactic instantiates V_{in}, V_{mid}, V_{out}. The tactic includes all variables given by the user (see above) and tries to add as few variables as possible to those sets in order to satisfy the "numerous conditions" from the Adversary rule.

Next, the tactic constructs a predicate R such that $(R \cap {\equiv}V_{out}) \subseteq B$. (Below we explain how R is constructed.) The updated postcondition is then defined to be $D := (R \cap {\equiv}V_{in})$.

Then, by rule Adversary, together with rule Seq and rule Conseq, we can replace the subgoal $\{A\}\mathbf{c}_0; \mathbf{c} \sim \mathbf{d}_0; \mathbf{d}\{B\}$ by the following subgoals:

- One subgoal ensuring some of the "numerous conditions". (Those that cannot be checked by the tactic internally.)
- One subgoal $\big\{R \cap {\equiv}V_{mid}\big\} s_i \sim s'_i \big\{R \cap {\equiv}V_{mid}\big\}$ for each mismatch s_i, s'_i.
- One subgoal $\{A\}\mathbf{c}_0 \sim \mathbf{d}_0\{D\}$.

Finally, we describe how R is computed:

- During the computation of $V_{in}, \dots$, a number of variables are identified that must not occur in R if the "numerous conditions" are to be satisfied. We write $Q_{forbidden}$ and $X_{forbidden}$ for those variables (the classical/quantum ones, respectively).
- Let V^{qu}_{out} denote the quantum variables in V_{out}. Then we remove the quantum equality $V^{\mathsf{qu}}_{out,1} \equiv_{\mathsf{quant}} V^{\mathsf{qu}}_{out,2}$ from B. That is, if $B = B' \cap \big(V^{\mathsf{qu}}_{out,1} \equiv_{\mathsf{quant}} V^{\mathsf{qu}}_{out,2}\big)$ (up to associativity and commutativity of $\cap$), we set $R_1 := B'$. If B cannot be parsed in this way, we set $R_1 := B$.

[13] The definition of multi-hole contexts is given in footnote 17 in [12].
Note that in statements of the form `call A(p1,...,pn)`, `A` is a context itself with `p1,...,pn` in its holes. So the holes of C can also be arguments of adversaries in `call`-statements. (E.g., when $\mathbf{c} =$ `call A(enc1)` and $\mathbf{d} =$ `call A(enc2)`, then $s_1 =$ `enc1` and $s'_1 =$ `enc2`.)

Obviously, $R_1 \cap \equiv V^{\mathtt{qu}}_{out} \subseteq B$, so R_1 is a candidate for the invariant R. Yet R_1 may still contain variables in $Q_{forbidden}$ and $X_{forbidden}$.

- If $Q_{forbidden} \cap fv(R) \neq \varnothing$, the tactic fails. (The variable sets $V_{in}, \dots$ are chosen above in a way that attempts that this does not happen, but it cannot be fully excluded.)
- Next, we remove all variables in $X_{forbidden}$ from R_1.
The simplest approach would be to set $R := \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2 \dots \mathbf{x}^{(n)}_1 \mathbf{x}^{(n)}_2} R_1$ where $\{\mathbf{x}^{(1)} \dots \mathbf{x}^{(n)}\} := X_{forbidden}$. (Essentially requiring that R_1 holds for any value of those variables.) Then we would have that $fv(R) \cap X_{forbidden} = \varnothing$. However, this approach is problematic because the resulting R may be too strong of an invariant. E.g., if $B = \mathfrak{Cla}[\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2]$, then R would be $R = \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2} \mathfrak{Cla}[\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2] = \mathfrak{Cla}\big[\forall_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2} \mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2\big] = \mathfrak{Cla}[\mathtt{false}]$.
Instead, let $\tilde{\mathbf{x}}^{(1)} \dots \tilde{\mathbf{x}}^{(m)}$ be all the classical variables in V_{out} such that for each i, both $\tilde{\mathbf{x}}^{(i)}_1$ and $\tilde{\mathbf{x}}^{(i)}_2$ occur in R. Let $R_2 := \mathfrak{Cla}\big[\neg(\tilde{\mathbf{x}}^{(1)} = \tilde{\mathbf{x}}^{(1)} \land \dots \land \tilde{\mathbf{x}}^{(m)} = \tilde{\mathbf{x}}^{(m)})\big] + R_1$. It is easy to verify that $R_2 \cap \equiv V^{\mathtt{qu}}_{out} \subseteq R_1$, and thus $R_2 \cap \equiv V^{\mathtt{qu}}_{out} \subseteq B$.
Finally, let $R := \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2 \dots \mathbf{x}^{(n)}_1 \mathbf{x}^{(n)}_2} R_2$. Then $R \cap \equiv V^{\mathtt{qu}}_{out} \subseteq B$ and $fv(R) \cap X_{forbidden} = \varnothing$.
And note that if, e.g., $B = \mathfrak{Cla}[\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2]$, then $R = \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2} \big(\mathfrak{Cla}[\neg(\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2)] + \mathfrak{Cla}[\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2]\big) = \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2} \mathfrak{Cla}[(\mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2) \implies \mathbf{x}^{(1)}_1 = \mathbf{x}^{(1)}_2] = \bigcap_{\mathbf{x}^{(1)}_1 \mathbf{x}^{(1)}_2} \mathfrak{Cla}[\mathtt{true}] = \mathfrak{Cla}[\mathtt{true}]$, avoiding the problem that the invariant becomes too strict.

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 if

The `if` tactic allows to replace an if-statement at the beginning of the left and/or right program by its then- or else-branch.

When invoked as "`if left`", it applies the following rule:

$$\frac{\{A \sqcap \mathfrak{Cla}[e_1]\}\mathbf{c}_{\mathtt{true}} \sim \mathbf{d}\{B\} \qquad \{A \sqcap \mathfrak{Cla}[\neg e_1]\}\mathbf{c}_{\mathtt{false}} \sim \mathbf{d}\{B\}}{\{A\}\mathbf{if}\ e\ \mathbf{then}\ \mathbf{c}_{\mathtt{true}}\ \mathbf{else}\ \mathbf{c}_{\mathtt{false}} \sim \mathbf{d}\{B\}}$$

That is, it splits the if-statement into two subgoals for each of the branches. Here e_1 is e with all variables $\mathbf{x}$ replaced by $\mathbf{x}_1$.

If we know that only the then-branch will be executed anyway, we can use "`if left true`" which applies the rule:

$$\frac{A \subseteq \mathfrak{Cla}[e_1] \qquad \{A \sqcap \mathfrak{Cla}[e_1]\}\mathbf{c}_{\mathtt{true}} \sim \mathbf{d}\{B\}}{\{A\}\mathbf{if}\ e\ \mathbf{then}\ \mathbf{c}_{\mathtt{true}}\ \mathbf{else}\ \mathbf{c}_{\mathtt{false}} \sim \mathbf{d}\{B\}}$$

Or if only the else-branch will be executed, we can use "`if left false`" which applies the rule:

$$\frac{A \subseteq \mathfrak{Cla}[\neg e_1] \qquad \{A \sqcap \mathfrak{Cla}[\neg e_1]\}\mathbf{c}_{\mathtt{false}} \sim \mathbf{d}\{B\}}{\{A\}\mathbf{if}\ e\ \mathbf{then}\ \mathbf{c}_{\mathtt{true}}\ \mathbf{else}\ \mathbf{c}_{\mathtt{false}} \sim \mathbf{d}\{B\}}$$

Furthermore, we can invoke the tactic as "`if right`", "`if right true`", or "`if right false`", with analogous behavior on the right program.

If both programs start with an if-statement, we can split both if-statements simultaneously using the `if joint` tactic. If is invoked as `if joint` l_1-r_1 $\dots l_n$-r_n where each l_i, r_i is a boolean (`true` or `false`). Each pair l_i-r_i stands for one of the possible combinations of value the left and right conditional can take. For example `if joint true-true false-false` means we claim that the left and right conditional will be equal. Then for each of these combinations, a subgoal is added containing the corresponding then- or else-branch in the left and right program. More precisely, the following rule is applied:

$$\frac{A \subseteq \mathfrak{Cla}[\exists i.\ e_1 = l_i \land f_2 = r_i] \qquad \text{for each } i\text{: } \{A \sqcap \mathfrak{Cla}[e_1 = l_i \land f_2 = r_i]\}\mathbf{c}_{l_i}; \mathbf{c}_{rest} \sim \mathbf{d}_{r_i}; \mathbf{d}_{rest}\{B\}}{\{A\}\mathbf{if}\ e\ \mathbf{then}\ \mathbf{c}_{\mathtt{true}}\ \mathbf{else}\ \mathbf{c}_{\mathtt{false}}; \mathbf{c}_{rest} \sim \mathbf{if}\ f\ \mathbf{then}\ \mathbf{d}_{\mathtt{true}}\ \mathbf{else}\ \mathbf{d}_{\mathtt{false}}; \mathbf{d}_{rest}\{B\}}$$

Here e_1 is e with all variables $\mathbf{x}$ replaced by $\mathbf{x}_1$ and f_2 analogously.

Note that the expression $\exists i.\ e_1 = l_i \land f_2 = r_i$ in first subgoal will not be stated in this precise form but in a logically equivalent one. (E.g., in case of the arguments `true-true false-false`, the expression is written $e_1 = f_2$.)

The common case `if joint true-true false-false` is the default when the tactic is invoked as `if joint`.

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 isa

When invoked as "`isa` M", it applies the Isabelle-method M to the first subgoal. For example, `isa simp` would be very similar to the builtin `simp` tactic. This is particularly useful to apply Isabelle methods that have no counterpart in the `qrhl-tool`.

For example, a particularly useful tactic for understanding why a certain ambient subgoal cannot be solved is to invoke `isa auto`. Since the `auto` method in Isabelle performs case distinctions, the resulting subgoals will often make it clearer what the remaining problem is than `simp` does.

When invoked as "`isa !` M", the tactic does the same thing but fails only the Isabelle method M completely solves the first subgoal.

Tactic local

The `local` tactic modifies the local variable declarations in a qRHL subgoal. It comes in several forms described below:

When invoked as `local remove left:` X (for some variables X) on a qRHL subgoal of the form $\{A\}$`{ local` Y`;` P_1 `}` $\sim P_2\{B\}$, it replaces the left program by $\{A\}$`{ local` $(Y \setminus X)$`;` P_1 `}` $\sim P_2\{B\}$. Each variable $v \in X$ must satisfy one of:

- v is not a free variable of P_1, or
- v_1 is not a free variable of A, B.

For classical variables, this requirement is checked by the tactic, and for quantum variables, a new subgoal is generated for this requirement (which can almost always be solved with `simp!`).

Analogously with `right` instead of `left`.

When invoked as `local remove left` or `local remove right` (i.e., without explicitly specified variables) it removes as many variables as possible.

This use of the tactic is justified by the RemoveLocal1/2 rules in [11].

When invoked as `local up` or `local up left` or `local up right`, it moves all local variable declarations in both, the left, or the right program upwards as far as possible. No additional subgoals are created.

When invoked as `local up` $v_1, \dots, v_n$ or `local up left` $v_1, \dots, v_n$ or `local up right` $v_1, \dots, v_n$, it moves the local variable declarations specified by $v_1, \dots, v_n$ upwards as far as possible. No additional subgoals are created. Each v_i is either a variable name (in which case all occurrences of `local` v_i are moved upwards), or v_i is of the form $v : i$, in which case the i-th occurrence of `local` v is moved upwards.

Tactic measure

When invoked as "measure.", converts a subgoal of the form $\{A\}\,\mathbf{c};\mathbf{x} <- \texttt{measure}\ Q\ \texttt{in}\ e \sim \mathbf{c}';\mathbf{x}' <- \texttt{measure}\ Q'\ \texttt{in}\ e'\,\{B\}$ (i.e., ending in a measurement-statement on both sides) into a subgoal $\{A\}\mathbf{c} \sim \mathbf{c}'\{C\}$ with suitably updated postcondition C.

Here e, e' must have the same type.

The tactic implements the following rule:

$$\frac{\{A\}\mathbf{c} \sim \mathbf{c}'\{B'\}}{\{A\}\,\mathbf{c};\mathbf{x}\ \texttt{<-}\ \texttt{measure}\ Q\ \texttt{in}\ e \sim \mathbf{c}';\mathbf{x}'\ \texttt{<-}\ \texttt{measure}\ Q'\ \texttt{in}\ e'\,\{B\}}$$

where

$$\begin{aligned} B' := \mathfrak{Cla}[e_1 = e_2'] \sqcap (Q_1 \equiv_{\mathsf{quant}} Q_2') \sqcap \\ \texttt{INF}\ z.\ \texttt{let}\ \bar e = ((\texttt{mproj}\ e_1\ z)\text{»}Q_1)\cdot\texttt{top};\ \bar e' = ((\texttt{mproj}\ e_2'\ z)\text{»}Q_2')\cdot\texttt{top}\ \texttt{in} \\ \big(B\{z/\mathbf{x}_1, z/\mathbf{x}_2'\} \sqcap \bar e \sqcap \bar f\big) + \texttt{ortho}\ \bar e + \texttt{ortho}\ \bar e' \end{aligned}$$

with

$$e_1 := \mathrm{idx}_1\, e, \quad e_2' := \mathrm{idx}_2\, e', \quad Q_1 := \mathrm{idx}_1\, Q, \quad Q_2' := \mathrm{idx}_2\, Q'.$$

This rule is a consequence of rule JointMeasureSimple and rule Seq: From rule JointMeasureSimple, we obtain $\{B'\}\,\mathbf{x}\ \texttt{<-}\ \texttt{measure}\ Q\ \texttt{in}\ e \sim \mathbf{x}'\ \texttt{<-}\ \texttt{measure}\ Q'\ \texttt{in}\ e'\,\{B\}$ (The only differences between B' and the precondition from rule JointMeasureSimple is the use of Isabelle-syntax here and the fact that $\mathrm{im}\, e$ replaced by the equivalent $e \cdot \texttt{top}$.) Then with $\{A\}\mathbf{c} \sim \mathbf{c}'\{B'\}$ and rule Seq, we get the conclusion of the rule.

Tactic o2h

This tactic allows to apply the Semiclassical O2H Theorem from [1] (Theorem 1, variant (1)). We refer to [1] for details about the O2H Theorem. To apply the O2H Theorem in `qrhl-tool`, we have the tactic `o2h`. As a precondition for applying this tactic, the games listed in Figure 2 must be defined. The games must be defined exactly as written there, except that the names of the games, as well as the names of the variables (IN, OUT, G, S, H, z, in_S, found, count) may be chosen arbitrarily. And `distr` can be an arbitrary constant expression (meaning, the expression must not contain any program variables but may contain ambient variables). Furthermore, we require that the type of the oracle outputs (i.e., β if G has type $\alpha \Rightarrow \beta$) is of type class `xor_group`,[14] otherwise `Uoracle` does not have the required behavior.

That is, `queryG` and `queryH` are implementations of the oracles that perform superposition queries to the functions G and H (using input/output registers IN, OUT). `Count` is a wrapper oracle that counts oracle queries (to express the bound on the number of queries performed by A). Let the programs `left`, `right` are just the programs defined in P_{left}, P_{right} in the O2H Theorem (see [1]). (Except that we additionally added a counter `count` that explicitly counts the oracle queries.) Finally, `queryGS` implements the "punctured oracle" $G \setminus S$ and stores in the variable `found` whether a value in S was queried. (A punctured oracle is one that allows superposition queries but measures whether the input register contains a value in S. In the definition of that program, "`binary_measurement (proj_classical_set S)`" constructs the binary measurement that checks this.) Thus the game `find` corresponds to P_{find} in the O2H Theorem.

Since the games have to be in this precise form, the first step before applying the tactic `o2h` will typically be to rewrite the games of interest in this specific form (for a suitably defined distribution `distr`) and show that the original and rewritten game have the same probability of $b = 1$.

The tactic `o2h` can then be applied to proof goals of the exact form:

```
abs ( Pr[b=1 : left(rho)] - Pr[b=1 : right(rho)] )
         <= 2 * sqrt( (1+real q) * Pr[found : find(rho)] )
```

where `left` and `right` are the games from Figure 2 and `q` is an expression (of type `nat`).

When applying the tactic `o2h` (without any additional arguments), it checks whether all involved games have the right form and that none of the variables `count,found,G,H,S,in_S` are in the free variables of A (but A is allowed to access `IN,OUT,b,z`). If these checks succeeds, the tactic produces four subgoals:

[14] This specifies an Abelian group with $x + x = 0$.

```
program queryG := {
  on IN, OUT apply (Uoracle G);
}.

program queryGS := {
  in_S <- measure IN with binary_measurement (proj_classical_set S);
  if (in_S=1) then found <- True; else skip;
  call queryG;
}.

program queryH := {
  on IN, OUT apply (Uoracle H);
}.

program Count(O) := {
  call O;
  count <- count + 1;
}.

program left := {
  count <- 0;
  (S,G,H,z) <$ distr;
  { local vars; call A(Count(queryG)); }
}.

program right := {
  count <- 0;
  (S,G,H,z) <$ distr;
  { local vars; call A(Count(queryH)); }
}.

program find := {
  count <- 0;
  (S,G,H,z) <$ distr;
  found <- False;
  { local vars; call A(Count(queryGS)); }
}.
```

Figure 2: Games required by `o2h` tactic. The local variable declaration `local vars` can be omitted (but then must be omitted in all games).

```
Pr[count ≤ q : left(rho)] = 1
Pr[count ≤ q : right(rho)] = 1
Pr[count ≤ q : find(rho)] = 1
∀S G H z x. (S,G,H,z) ∈ supp distr → x ∉ S → G x = H x
```

The first three of them express the requirement that A makes at most q oracle queries (recall that `count` counts the oracle queries because of the wrapper oracle `Count`). And the fourth one expresses the fact that $\forall x \notin S, G(x) = H(x)$ when S, G, H are chosen according to `distr`. (This is one of the premises of the O2H Theorem.)

Note that the program `find` contains a punctured oracle `queryGS`. To transform `find` into a game with normal oracles (such as `queryG`), see the tactic `semiclassical`.

Tactic rename

When invoked as `rename left:` σ or `rename right:` σ or `rename both:` σ, renames free variables in the left/right/both programs according to the substitution σ. σ must be specified as a sequence of mappings of the form `a->b, c->d, e->f, ...`.

Assume the current subgoal is $\{A\}\mathbf{c} \sim \mathbf{d}\{B\}$.

For the tactic to be applicable, the following conditions must be satisfied:

- The variables in the domain D of the substitution (i.e., `a,c,e,...`) have to be distinct.
- The variables in the range R of the substitution (i.e., `a,c,e,...`) have to be distinct.
- For each mapping `a->b` in the substitution, the variables must have the same type, and must be both classical or both quantum.
- Applying σ to the left/right/both programs must no lead to a collision between local and renamed free variables. (Formally, $\mathit{noconflict}(\sigma, \mathbf{c})$ and/or $\mathit{noconflict}(\sigma, \mathbf{d})$ must hold where *noconflict* is defined in [11].)
- $R \setminus D \cap (\mathit{fv}(\mathbf{c}) \cup V_{A1} \cup V_{B1}) = \varnothing$ where $V_{A1} := \{\mathbf{x} : \mathbf{x}_1 \in \mathit{fv}(A)\}$ and $V_{B1} := \{\mathbf{x} : \mathbf{x}_1 \in \mathit{fv}(B)\}$ (free variables of A, B with index 1 removed). (Only in cases `left` and `both`.)
- $R \setminus D \cap (\mathit{fv}(\mathbf{d}) \cup V_{A2} \cup V_{B2}) = \varnothing$ where $V_{A2} := \{\mathbf{x} : \mathbf{x}_2 \in \mathit{fv}(A)\}$ and $V_{B2} := \{\mathbf{x} : \mathbf{x}_2 \in \mathit{fv}(B)\}$ (free variables of A, B with index 2 removed). (Only in cases `right` and `both`.)
- $R^{\mathsf{qu}} \cap (V_{A1} \cup V_{B1}) = \varnothing$ in cases `left` and `both`, and $R^{\mathsf{qu}} \cap (V_{A2} \cup V_{B2}) = \varnothing$ in cases `right` and `both`. (Here R^{qu} are the quantum variables in R.)[15]
- Renaming $\mathbf{c}\sigma$ and/or $\mathbf{d}\sigma$ (depending on `left/right/both`) must be possible without renaming a variable inside a declared program (included via a `call`-statement).[16]

The tactic creates one or two subgoals:

- A subgoal that checks some of the above variable conditions. (This subgoal may be missing if the tactic can check everything internally.)
- $\{A\sigma_1\}\mathbf{c}\sigma \sim \mathbf{d}\{B\sigma_1\}$ or $\{A\sigma_2\}\mathbf{c} \sim \mathbf{d}\sigma\{B\sigma_2\}$ or $\{A\sigma_1\sigma_2\}\mathbf{c}\sigma \sim \mathbf{d}\sigma\{B\sigma_1\sigma_2\}$ (in cases `left`, `right`, `both`).
  Here σ_1, σ_2 are the substitutions that rename the 1-indexed/2-indexed variables according to sigma. (I.e., σ_1 renames `a1->b1, c1->d1, e1->f1, ...` and σ_2 renames `a2->b2, c2->d2, e2->f2, ...`)

The tactic is justified by rule RenameQrhl1/2 in [11].

Tactic rnd

Converts a subgoal of the form $\{A\}\ \mathbf{c}; \mathbf{x}$ `<$` $e \sim \mathbf{c}'; \mathbf{x}'$ `<$` $e'\ \{B\}$ (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:

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

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

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

[15] This condition is not required by rule RenameQrhl1/2. However, the tactic requires it because of the way how the renaming of quantum variables is computed internally.

[16] This condition is not required by rule RenameQrhl1/2. However, if it is not satisfied, the result of renaming cannot be expressed without renaming the declared programs.

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 "$\texttt{rnd } \mathbf{x}, \mathbf{x}' \texttt{ <- } f\texttt{.}$" implements the following rule:

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

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

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 $\texttt{rnd}\ F\ G$ where F and G are isomorphisms between the distributions e_1, e_2'. The EasyCrypt behavior can be recovered in our tool by invoking $\texttt{rnd } \mathbf{x}, \mathbf{x}' \texttt{ <- map\_distr } (\lambda z.\, (z, F\, z))\ e_1$. (Instead of the condition that F is an isomorphism between the distributions, our tactic will have the equation $\texttt{map\_distr snd map\_distr } (\lambda z.\, (z, F\, z))\ e_1 = e_2'$ 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 $\{\texttt{top}\}\mathbf{x} \texttt{ <\$ } d; \sim \mathbf{x} \texttt{ <\$ map\_distr } (\lambda z.\ z * z)\ d; \{\mathfrak{Cla}[\mathbf{x}_1 * \mathbf{x}_1 = \mathbf{x}_2]\}$ (see the contributed file `rnd.qrhl`[17]) which does not seem easily possible in EasyCrypt.

Tactic rule

When invoked as "$\texttt{rule}\ l$" 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 $A_1\sigma$, ..., $A_n\sigma$.

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 $\texttt{distinct\_qvars } [\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$ that will then become new subgoals in the tool and can be resolved using the `simp`-tactic).

l can be any specification of a lemma that Isabelle understands. That is, we can also write, e.g., $f(3)$ for the third part of the fact f, or $f[\texttt{where } x = 1]$ to instantiate x with 1 in f, etc.

Tactic semiclassical

The tactic `o2h` above introduces games that contain "punctured oracles", i.e., oracles that allow superposition queries to a function but measure whether the input is in a given set S. At some point, it is usually necessary to get rid of the punctured oracle. Theorem 2 in [1] gives a method to do so. The tactic `semiclassical` (invoked without any arguments) implements that theorem. This tactic requires that games of the exact form as in Figure 3 are defined. (The names of the games, as well as the variables (IN, OUT, G, S, H, z, in_S, found, count, stop_at, guess) can be arbitrary, and the output type of G must be of type class `xor_group`. `distr` and `q` are arbitrary constant expressions.) See the description of the tactic `o2h` for programs `queryG`, `queryGS`, `Count`. The program queryGM is an oracle that first checks whether the number of the current oracle query is `stop_at` before querying G. If so, the input to G is measured in the computational basis and stored in `guess`. This corresponds to the query performed by the adversary B in Theorem 2in [1]. (Where `stop_at` is i in B.) And finally, the game `left` is like the `find` game in tactic `o2h`.

Then the tactic `semiclassical`, invoked without any arguments, expects a subgoal of the form:

```
Pr[found : left(rho)]
          <= 4 * real q * Pr[guess∈S : right(rho)]
```

It checks whether all games are as in Figure 3 and whether the free variables of A contain none of `G, S, H, in\_S, found, count, stop\_at, guess` (but A may access `IN, OUT, z, b`). If so, the tactic produces the following new subgoals:

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

```
program queryG := {
  on IN, OUT apply (Uoracle G);
}.

program queryGS := {
  in_S <- measure IN with binary_measurement (proj_classical_set S);
  if (in_S=1) then found <- True; else skip;
  call queryG;
}.

program queryGM := {
  if (count=stop_at) then
    guess <- measure IN with computational_basis;
  else
    skip;

  call queryG;
}.

program Count(O) := {
  call O;
  count <- count + 1;
}.

program left := {
  count <- 0;
  (S,G,z) <$ distr;
  found <- False;
  { local vars; call A(Count(queryGS)); }
}.


program right := {
  count <- 0;
  stop_at <$ uniform {..<q};
  (S,G,z) <$ distr;
  { local vars; call A(Count(queryGM)); }
}.
```

Figure 3: Games required by `semiclassical` tactic. The local variable declaration `local vars` can be omitted (but then must be omitted in all games).

```
Pr[count ≤ q : left(rho)] = 1
Pr[count ≤ q : right(rho)] = 1
```

Here `q` is the same expression as in the definition of program `queryGM` and `right`. These subgoals express the fact that the adversary makes at most q oracle queries.

Tactic seq

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

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

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 $P_1 \implies \cdots \implies P_n \implies \{A\}\mathbf{c} \sim \mathbf{d}\{B\}$).

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 $l_1, \dots, l_n$ 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 squash

When invoked as `squash left` (or `squash right`) on a qRHL subgoal, it replaces the last two assign/sample statements $\mathbf{c}_1; \mathbf{c}_2$ in the left (or right) subgoal by a single assign/sample statement $\mathbf{c}'$ with the same effect. If the last two statements are not assign/sample statements, the tactic fails.

This tactic is useful, e.g., when we want to use the `rnd` tactic but in one program variables $\mathbf{x}$ and $\mathbf{y}$ are sampled in two separate statements while on the other side they are sampled simultaneously using a joint distribution. Then we can join the two statements in the first program using `squash` before using `rnd`.

We distinguish four cases. In the following X, Y are tuples of variables, d, e are expressions. And $e'\ z := e\{z/X\}$.

- $\mathbf{c}_1 = X \stackrel{\$}{\leftarrow} d$ and $\mathbf{c}_2 = Y \stackrel{\$}{\leftarrow} e$:
  Then $\mathbf{c}' = (X, Y) \stackrel{\$}{\leftarrow} \texttt{bind\_distr}\ d\ (\lambda z.\ \texttt{map\_distr}\ (\lambda y.\ (z, y))\ (e'\ z))$.
- $\mathbf{c}_1 = X \leftarrow d$ and $\mathbf{c}_2 = Y \stackrel{\$}{\leftarrow} e$:
  Then $\mathbf{c}' = (X, Y) \stackrel{\$}{\leftarrow} \texttt{bind\_distr}\ (\texttt{point\_distr}\ d)\ (\lambda z.\ \texttt{map\_distr}\ (\lambda y.\ (z, y))\ (e'\ z))$.
- $\mathbf{c}_1 = X \stackrel{\$}{\leftarrow} d$ and $\mathbf{c}_2 = Y \leftarrow e$: Then $\mathbf{c}' = (X, Y) \stackrel{\$}{\leftarrow} \texttt{map\_distr}\ (\lambda y.\ (y, e\ y))\ (e'\ z)$.
- $\mathbf{c}_1 = X \leftarrow d$ and $\mathbf{c}_2 = Y \leftarrow e$: Then $\mathbf{c}' = (X, Y) \leftarrow (d, e'd)$.

Note that this is allowed even if X and Y share variables.[18]

Tactic swap

When invoked as `swap left` *`range`* n (or `swap left` *`range`* n) on a qRHL subgoal, it moves the block of lines identified by the range r in the left (or right) program forward by n lines. Here r is either for the form "a–b" (meaning lines a till b) or "a" (meaning the last a lines).

The tactic can only be applied if the following condition is satisfied: Let $\mathbf{c}$ denote the program fragment to be moved, and $\mathbf{d}$ the n lines before $\mathbf{c}$.

- The quantum variables of $\mathbf{c}$ and $\mathbf{d}$ must be disjoint.
- The written classical variables[19] of $\mathbf{c}$ must be disjoint from the classical variables of $\mathbf{d}$.

[18] This works because if X and Y share a variable $\mathbf{x}$, then in (X, Y), the rightmost occurrence of $\mathbf{x}$ determines what is assigned to $\mathbf{x}$, which is the intended behavior in this case.

[19] Written classical variables are those on the lhs of an assign/sample/measurement statement that are not hidden under a `local` statement.

- The written classical variables of $\mathbf{d}$ must be disjoint from the classical variables of $\mathbf{c}$.

Tactic wp

Removes the last statement(s) from the left or right program of a qRHL subgoal and adapts the post-condition accordingly.

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

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

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:

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

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,[20] Measure1, QInit1,[21] 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.)

`wp left` and `wp right` apply only to the very last statement. The following variants can be used to handle several statements in one go: `wp left` n with $n \geq 0$ is equivalent to n invocations of `wp left`. Analogously `wp right` n. And `wp` n m is equivalent to `wp left` n`.` `wp right` n`.`

[20]Note that rule QApply1 does not contain the term $\mathfrak{Cla}[\mathtt{isometry}\ e_1]$ that $\mathrm{wp}_1(B, \mathtt{on}\ \mathbf{q}^{(1)},\dots,\mathbf{q}^{(n)}\ \mathtt{apply}\ e)$ contains. The reason why $\mathrm{wp}_1(\dots)$ includes this additional term is that $\mathtt{on}\ \mathbf{q}^{(1)},\dots,\mathbf{q}^{(n)}\ \mathtt{apply}\ e$ actually translates to **apply** $\mathit{mkIso}(e)$ **to** $q_1,\dots,q_n$ (see footnote 9). Applying rule QApply1 to this program gives the precondition

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

which is a superset of

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

[21]Note that rule QInit1 does not contain the term $\mathfrak{Cla}[\mathtt{norm}\ e_1 = 1]$ that $\mathrm{wp}_1(B,\ \mathbf{q}^{(1)},\dots,\mathbf{q}^{(n)}\ \mathtt{\texttt{<}q}\ e)$ contains. The reason why $\mathrm{wp}_1(\dots)$ includes this additional term is that $\mathbf{q}^{(1)},\dots,\mathbf{q}^{(n)}\ \mathtt{\texttt{<}q}\ e$ actually translates to $\mathbf{q}_1,\dots,\mathbf{q}_n \overset{\mathbf{q}}{\leftarrow} \mathit{mkUnit}(e)$ (see footnote 8). Applying rule QInit1 to this program gives the precondition

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

which is a superset of

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

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 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 [7] and the reference manual [14] for more information. The theory `QRHL` (imported using `imports QRHL.QRHL` in Isabelle) 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 [14]. Or try the `sledgehammer` command [14] 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 [12]**
`leq_space_div` $^{\text{simp}}$	Lemma 21
`classical_inf` $^{\text{simp}}$ `classical_sup` $^{\text{simp}}$ `Cla_plus` $^{\text{simp}}$ `BINF_Cla` $^{\text{simp}}$ `classical_ortho` $^{\text{simp}}$	Lemma 25
`qeq_collect` `qeq_collect_guarded` $^{\text{simp}}$	Lemma 31
`Qeq_mult1` `Qeq_mult2`	Lemma 32
`quantum_eq_unique` $^{\text{simp}}$	Lemma 33
`quantum_eq_add_state`	Lemma 34
$^{\text{simp}}$ 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).

6.1 Declaring types

In an accompanying Isabelle file, it is possible to define types as usual using Isabelle commands such as `typedef`, `datatype`, and `typedecl`. However, there is one important caveat: To use a type as the type of a program variable, that type needs to instantiate the type class `universe` (representing types of sufficiently small cardinality).[22] For most builtin types, this is already the case. However, there are two cases where one needs to be aware of this restriction.

First, when defining one's own types (using `typedef` or `datatype`). In that case, Isabelle will not know that the resulting type is small. Fortunately, in most cases (assuming the types from which the new type is built are small) this can be done automatically with a single command:

```
derive universe typename
```

For example:

```
datatype 'a mytree = Node "'a mytree * 'a mytree" | Leaf 'a .
derive universe mytree
```

The `derive universe` command is also useful for types imported from other Isabelle theories if they were not yet shown to instantiate `universe`.

The second use case is the declaration of types using `typedecl`. Such declarations are useful to specify in a development that `T` is just an arbitrary type, and that the whole proof holds for any type `T`. (E.g., one might declare a type `key` of keys without further specifying its nature.) However, in this case,

[22] For reasons described in footnote 4.

Isabelle will not know that T is small (and thus instantiates universe) since nothing was specified about T. One solution would be to add instantiation proofs with sorry. However, we have included a custom command for declaring types that covers this situation: declare_variable_type. In its basic form, it declares a new type that is of class universe:

```
declare_variable_type key
```

The new type can also have type parameters:

```
declare_variable_type 'a t1
declare_variable_type ('a,'b) t2
declare_variable_type ('a::finite,'b) t3
```

where the last case constrains 'a to have the type class finite. Finally, the command also has a convenient method for declaring that a given type has further type classes (besides universe) such as in:

```
declare_variable_type key :: finite
declare_variable_type msg :: "{finite,xor_group}"
declare_variable_type ('a::finite) list :: finite
```

(universe is always implicitly added.[23])

The command will check whether the existence of a type of the given sorts can be consistently assumed. If this is not the case, a warning is issued.[24] For example,

```
declare_variable_type wrong :: "{finite,no_top}"
```

produces a warning since it would declare a type that is both finite and has no upper bound, and thus lead to a contradiction.

6.2 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 [4]. 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 $A, B :: (\alpha, \beta)$ cblinfun.) In our case, we give algorithms for most operations on bounded operators and subspaces. (We rely heavily on [9] 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.[25]

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

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

Here the lhs and rhs are infinite dimensional subspaces (because $\text{»}[\![\mathbf{q}_1, \mathbf{q}_2]\!]$ 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: $\texttt{span}\ \{\mathtt{EPR}\} \leq \texttt{span}\ \{|00\rangle, |11\rangle\}$.

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 lifts can be interspersed with different operations, and they may use different sets of quantum variables, or differently ordered ones. For example, consider

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

[23]More precisely, declare_variable_type ('$\mathtt{a}_1$::$\mathtt{s}_1$,...,'$\mathtt{a}_n$::$\mathtt{s}_n$) t :: s declares two facts: If '$\mathtt{a}_1$,...,'$\mathtt{a}_n$ have sorts (type classes) $\mathtt{s}_1$,...,$\mathtt{s}_n$, then the type ('$\mathtt{a}_1$,...,'$\mathtt{a}_n$) t has sort s. (Where $\mathtt{s}_i$ and s are empty when omitted from the command.) And if '$\mathtt{a}_1$,...,'$\mathtt{a}_n$ have sorts $\mathtt{s}'_1$,...,$\mathtt{s}'_n$, then the type ('$\mathtt{a}_1$,...,'$\mathtt{a}_n$) t has sort $\mathtt{s}'$, where $\mathtt{s}'_i$ and $\mathtt{s}'$ are $\mathtt{s}_i$ and s with the type class universe added.

[24]If the warning is not justified, it is possible to remove it by manually defining a new type (e.g., via typedef or datatype) and showing that that type has the required sort (it is a *sort witness*). After that, declare_variable_type will not issue a warning any more since the existence of a type of the right sort is ensured.

[25]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.

(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 $\texttt{span}\ \{|00\rangle, |11\rangle\}»[\![\mathbf{q}_2, \mathbf{q}_1]\!]$ into $\big(\texttt{comm\_op} \cdot \texttt{span}\ \{|00\rangle, |11\rangle\}\big)»[\![\mathbf{q}_1, \mathbf{q}_2]\!]$ (where comm_op is an operator mapping $|x, y\rangle$ to $|y, x\rangle$), and only then can we apply the rule $A \leq B \implies A»Q \leq B»Q$ 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.[26] (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 lifts.)

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

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

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

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

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

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

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

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, $\mathrm{Adv}^{A_1 A_2}_{\mathrm{ROR}}$ is negligible." This is what is called asymptotic security. We will instead follow the concrete security approach where we explicitly derive bounds for $\mathrm{Adv}^{A_1 A_2}_{\mathrm{ROR}}$.

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`.[27]

In this theory, we first declare the types `key` and `msg` as abstract (i.e., unspecified) types. We want both types to be finite, i.e., of type class `finite` (otherwise uniform sampling of keys/messages is not well-defined), and we want that on type `msg`, $+$ represents the XOR operation (type class `xor_group`[28]).

```
declare_variable_type key :: finite
declare_variable_type msg :: "{finite,xor_group}"
```

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)"
```

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`.[29] We first specify the games from Definition 1

[27] 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`.

[28] This type class declares `msg` as an abelian additive group with the extra law $a + a = 0$.

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

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

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 4. 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 $\{\dots\}\dots;$ `call` $A \sim \dots;$ `call` $A\{\dots\}$, 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 $\{\dots\}\dots;$ `k <$ uniform UNIV` $\sim \dots;$ `s <$ uniform UNIV` $\{\dots\}$, 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

$$\{\dots\}\dots;\ \texttt{r <\$ uniform UNIV;} \sim \dots;\ \texttt{r <\$ uniform UNIV;}\{\ \mathfrak{Cla}[\texttt{G k1 + r1 = r2 + m2} \land \texttt{b1 = b2} \land \texttt{cglobA1 = cglobA2}] \sqcap [\![\texttt{qglobA1}]\!] \equiv \texttt{q}\ [\![\texttt{qglobA2}]\!]\ \}$$

At 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

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

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.

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,[30] we have to describe the two IND-OT-CPA games from Definition 4 (`indcpa0` and `indcpa1` in Figure 5), 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

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

an adversary module), we need to write down the games from Definition 2 twice, once for adversary B_1 (`prg0B1` and `prg1B1` in Figure 5) 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:

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

Here $\xleftrightarrow{=}$ means that we show that the probability of $b = 1$ is the same in the two games. And $\xleftrightarrow{\mathrm{Adv}^{B_i}_{\mathrm{PRG}}}$ 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 $\xleftrightarrow{=}$ 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`.

7.3 Quantum equality

In the file `equality.qrhl`[31] we give a simple example involving reasoning about quantum equality. We show

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

[30] 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`.

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

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

$\{I_1\}$ `b <$ uniform UNIV; if (b=1) then on q apply hadamard; else skip;`
$\sim$ `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

$\{I_2\}$ `if (b=1) then on q apply hadamard; else skip;`
$\sim$ `on q apply (if b=1 then hadamard else idOp);` $\{\mathbf{q}_1 \equiv_{\mathsf{quant}} \mathbf{q}_2\}$ (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 $\mathfrak{Cla}$[`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

```
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;  }.
```

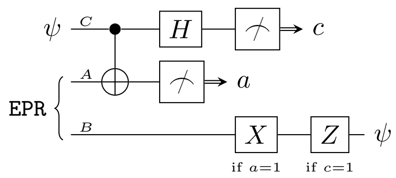

Figure 6: Quantum teleportation as a program and as a circuit.

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 [3]. 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 6. We will show the following fact:

$$\{C_1 \equiv_{\mathsf{quant}} A_2\}\texttt{teleport} \sim \mathbf{skip}\{B_1 \equiv_{\mathsf{quant}} A_2\} \tag{7}$$

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`.[32] 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 $I_1 := (C_1 \equiv_{\mathsf{quant}} A_2) \sqcap (\texttt{span}\{\texttt{EPR}\}\text{»}[\![A_1, B_1]\!])$ 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 $\{\texttt{EPR}\}$. 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 34 in [12]). Intuitively, I_2 states that after the initialization, $C_1 A_1 B_1$ 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 $C_1 A_1 B_1$ 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`,[33] 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 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 the theory `QRHL`. 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:

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

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

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

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 $I_4 := \texttt{Proj (span \{ket } a_1\texttt{\})»}[\![A_1]\!] \cdot I_3$ 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:

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

We remove the all-quantifier using tactic `fix a'`. Then the fact can be shown using tactic `rule move_plus_meas_rule`,[34] 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 $I_5 := \mathfrak{Cla}[a_1 = a'] \sqcap \texttt{Proj (span \{ket } c_1\texttt{\})»}[\![C_1]\!] \cdot I_4$ 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

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

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 $B_1 \equiv_{\mathsf{quant}} A_2$. 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.

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

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`.[35] The definition of the program `teleport` is the same as before (see Figure 6). 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 $\{\dots\}\mathbf{skip} \sim \mathbf{skip}\{\dots\}$, 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`[36] 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] Andris Ambainis, Mike Hamburg, and Dominique Unruh. "Quantum Security Proofs Using Semi-classical Oracles". In: *Crypto 2019*. Springer, 2019, pp. 269–295.

[2] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. *CryptHOL: Game-based Proofs in Higher-order Logic*. IACR ePrint 2017/753. 2017.

[3] 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`.

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

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

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

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

[8] The PG dev team. *Proof General (A generic Emacs interface for proof assistants)*. `https://proofgeneral.github.io/`. Accessed 2018-10-24.

[9] 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.

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

[11] Dominique Unruh. *Local Variables and Quantum Relational Hoare Logic*. arXiv:2007.14155 [cs.LO]. 2020.

[12] Dominique Unruh. *Quantum Relational Hoare Logic*. arXiv:1802.03188v2 [quant-ph]. Published at POPL as [13]. 2019.

[13] Dominique Unruh. "Quantum relational Hoare logic". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). Full version is [12], 33:1–33:31. ISSN: 2475-1421. DOI: `10.1145/3290346`. URL: `http://doi.acm.org/10.1145/3290346`.

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

Symbol index

$\ell^2[V]$	$\ell^2(\mathsf{Type}_V^{\mathsf{set}})$ – Hilbert space with basis $\mathsf{Type}_V^{\mathsf{set}}$
$\mathbf{D}(B)$	Distributions on B
$\mathbf{D}[V]$	$\mathbf{D}(\mathsf{Type}_V^{\mathsf{set}})$ – Distributions on $\mathsf{Type}_V^{\mathsf{set}}$

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

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

$\mathbf{Iso}(X,Y), \mathbf{Iso}(X)$	Isometries from $\ell^2(X)$ to $\ell^2(Y)$ (on $\ell^2(X)$)	
$\mathbf{apply}\ \mathbf{q}_1 \dots \mathbf{q}_n\ \mathbf{to}\ U$	Statement: Apply unitary/isometry U to quantum registers $\mathbf{q}_1 \dots \mathbf{q}_n$	
$\mathbf{x} \leftarrow \mathbf{measure}\ \mathbf{q}_1 \dots \mathbf{q}_n\ \mathbf{with}\ e$	Statement: Measure quantum variables $\mathbf{q}_1 \dots \mathbf{q}_n$ with measurement e	
$\mathbb{B}$	Booleans. $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$	
$\ell^2(B)$	Hilbert space with basis indexed by B	
$\mathbf{Iso}[V,W], \mathbf{Iso}(V)$	Isometries from $\ell^2[V]$ to $\ell^2[W]$ (on $\ell^2[V]$)	
$\mathbf{U}(X,Y), \mathbf{U}(X)$	Unitaries from $\ell^2(X)$ to $\ell^2(Y)$ (on $\ell^2(X)$)	
$fv(e)$	Free variables in an expression e (or program)	
$\mathsf{Type}^{\mathsf{list}}_V$	Type of a list V of variables	
$A \mathbin{\Delta} B$	Symmetric difference of sets	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{nonsep}}$	qRHL judgment, non-separable definition	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}$	Quantum relational Hoare judgment	
id	Identity	
id_V	Identity on $\ell^2[V]$ or on $\mathbf{B}[V]$	
$\lvert b\rangle, \lvert b\rangle_V$	Basis vector in Hilbert space $\ell^2[V]$	
A^*	Adjoint of the operator A	
$\mathbf{B}(X,Y)$	Bounded linear operators from $\ell^2(X)$ to $\ell^2(Y)$	
$\mathcal{E}$	A superoperator	
$\mathbf{B}[V,W]$	Bounded linear operators from $\ell^2[V]$ to $\ell^2(W)$	
$\mathbf{B}^{\leq 1}(X,Y)$	Bounded linear operators with operator norm ≤ 1	
$a \circ_e b$	Composition of expressions a, b as relations	
`top`	Full subspace (Isabelle/HOL syntax)	10
$U_{vars,Q}$	Canonical isomorphism between $\ell^2(\mathsf{Type}^{\mathsf{list}}_Q)$ and $\ell^2[\mathsf{Type}^{\mathsf{set}}_Q]$ for a list Q	
$A \div \psi$	Part of A containing ψ	
$A \sqcup B$	Sum of subspaces (Isabelle/HOL syntax)	10
$A \sqcap B$	Intersection of subspaces (Isabelle/HOL syntax)	10
`bot`	Zero subspace (Isabelle/HOL syntax)	10
`INF x:Z. e`	Intersection of family of subspaces (Isabelle/HOL syntax)	10
$\operatorname{supp} \mu$	Support of distribution μ	
$\mathbf{supp}\, M$	Support of an operator M	
$f(x := y)$	Function update, i.e., $(f(x := y))(x) = y$	
$x \leftarrow e$	Program: assigns expression e to x	
$\mathrm{idx}_1\, \mathbf{c}, \mathrm{idx}_1\, e$	Add index 1 to every variables in $\mathbf{c}$ or e	
$\mathbf{Meas}(D,E)$	Projective measurements on $\ell^2(E)$ with outcomes in D	
$\mathsf{marginal}_i(\mu)$	i-th marginal distribution of μ (for $\mu \in \mathbf{D}^{\leq 1}(X \times Y)$, $i = 1, 2$)	
$S^\perp$	Orthogonal complement of subspace S	
V^{cl}	Classical variables in V	
$\mathsf{Type}^{\mathsf{set}}_X$	Type of a set V of variables	
$\mathrm{Adv}^A_{\mathrm{PRG}}(\eta)$	Advantage of adversary A in PRG-OT-CPA game with security parameter η	33
$\mathrm{Time}(A)$	Worst-case runtime of A	33
$[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$	Typed tuple of quantum variables (Isabelle/HOL syntax)	14
$P \div \psi\text{»}Q$	Isabelle/HOL syntax for `space_div` (related to $P \div \psi$)	17
$Q_1 \equiv\!\mathtt{q}\ Q_2,\ Q_1 \mathtt{==q}\ Q_2$	Isabelle/HOL syntax for quantum equality $Q_1 \equiv_{\mathsf{quant}} Q_2$	16
$\mathtt{Qeq}[\mathbf{q}_1, \dots, \mathbf{q}_{\mathtt{n}} = \mathbf{q}'_1, \dots, \mathbf{q}'_{\mathtt{m}}]$	Isabelle/HOL syntax for quantum equality $\mathbf{q}_1, \dots, \mathbf{q}_n \equiv_{\mathsf{quant}} \mathbf{q}'_1, \dots, \mathbf{q}'_m$	16
$\mathrm{Adv}^{A_1 A_2}_{\mathrm{ROR}}(\eta)$	Advantage of adversary A_1, A_2 in ROR-OT-CPA game with security parameter η	32

$\mathrm{Adv}^{A_1 A_2}_{\mathrm{IND}}(\eta)$	Advantage of adversary A_1, A_2 in IND-OT-CPA game with security parameter η	35
$\lfloor x \rfloor$	x rounded down to the next integer	
$x \in_{\$} M$	x uniformly sampled from M	32
$\mathbf{T}[V]$	Trace class operators on $\ell^2[V]$	
$A \otimes B$	Tensor product of vectors/operators A and B	
$\mathbb{C}$	Complex numbers	
$\mathbf{D}^{\leq 1}[V]$	Sub-probability distributions over variables V	
$\mathbf{D}^{\leq 1}(X)$	Sub-probability distributions over X	
$\mathbf{U}[V, W], \mathbf{U}(V)$	Unitaries from $\ell^2[V]$ to $\ell^2[W]$ (on $\ell^2[V]$)	
$\mathbf{T}^+(X)$	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	
$\mathbb{R}$	Real numbers	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{uniform}}$	qRHL judgment, uniform definition	
$\{F\}\mathbf{c} \sim \mathbf{c}'\{G\}_{\mathsf{class}}$	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	
$\lVert x \rVert$	ℓ_2-norm of vector x, or operator-norm	
$\mathrm{tr}\, M$	Trace of matrix/operator M	
$\mathfrak{Cla}[e]$	Classical predicate meaning $e = \mathtt{true}$	
$\mathrm{span}\, A$	Span, smallest subspace containing A	
$X_1 \equiv_{\mathsf{quant}} X_2$	Equality of quantum variables X_1 and X_2	
V^{qu}	Quantum variables in V	
Type_v	Type of variable v	
$A \gg Q$	Overloaded Isabelle/HOL constant for `liftOp`, `liftSpace`	14, 15
$[\![\mathbf{q}_1, \dots, \mathbf{q}_n]\!]$	Typed tuple of quantum variables (Isabelle/HOL syntax)	
2^M	Powerset of M	
$m_1 m_2$	Union (concatenation) of memories m_1, m_2	
x `<-` e`;`	Assignment (tool program syntax)	7
$\otimes$	Tensor product (Isabelle/HOL constant)	13, 14, 17
$\mathbf{q}_1, \dots, \mathbf{q}_n$ `<q` e`;`	Quantum initialization (tool program syntax)	7
x `<$` e`;`	Sampling (tool program syntax)	7
$A \cdot B$	Overloaded Isabelle/HOL constant for `timesOp`, `cblinfun_apply`, `applyOpSpace`	12, 12, 13
`Pr`$[v : P(\rho)]$	Isabelle/HOL constant for probability of $v = 1$ after running P	12
$\mathbf{c}$	A program	
$[\![e]\!]_m$	Denotation of a classical expression e, evaluated on classical memory m	
$[\![\mathbf{c}]\!]$	Denotation of a program $\mathbf{c}$	
$\mathbf{d}$	A program	
skip	Program that does nothing	
$[\![\mathbf{c}]\!]_{\mathbf{class}}$	Classical denotation of a program $\mathbf{c}$	
while e **do** c	Statement: While loop	
if e **then** c_1 **else** c_2	Statement: If (conditional)	
$\mathbf{q}_1 \dots \mathbf{q}_n \overset{\mathbf{q}}{\leftarrow} e$	Statement: Initialize $\mathbf{q}_1, \dots, \mathbf{q}_n$ with quantum state e	
$\mathbf{x} \overset{\$}{\leftarrow} e$	Statement: Sample $\mathbf{x}$ according to distribution e	
$\mathbb{N}$	Natural numbers	
$R_1 \circ R_2$	Composition of relations	

$\mathsf{lift}(\mu)$	Transforms a distribution μ into a density operator
$A»Q$	Lifts operator or subspace to variables Q
`false`	Truth value "false"
$\mathbf{T}^+_{cq}[V]$	Positive trace class cq-operators on $\ell^2[V]$
$\mathbf{c};\mathbf{d}$	Sequential composition of programs
`true`	Truth value "true"
δ_x	Point distribution: returns x with probability 1
$\downarrow_e(\rho)$	Restrict state/distribution ρ to the case $e = \mathtt{true}$ holds
$\mathbf{T}^+[V]$	Positive trace class operators on $\ell^2[V]$
$\mathbf{T}_{cq}[V]$	Trace class cq-operators on $\ell^2[V]$
$\|x\|$	Absolute value of x / cardinality of set x
$\mathit{overwr}\,\mathbf{c}$	Overwritten variables in program $\mathbf{c}$
$\mathsf{Type}^{\mathsf{exp}}_e$	Type of an expression e
$\mathbf{x}$	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
$\mathsf{proj}(x)$	Projector onto x, i.e., xx^*
$\mathbf{q}$	A quantum program variable
$\mathbf{y}$	A classical program variable

Index