Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 338 (2018) 185–201

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formal Meta-level Analysis Framework for Quantum Programming Languages

Mohamed Yousri Mahmoud[1] Amy P. Felty[2]

School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada

Abstract

The design and development of quantum programming languages (QPLs) is an important and active area of quantum computing. This paper addresses the problem of developing a standard methodology for verifying a QPL against major quantum computing concepts. We propose a framework dedicated to the meta-level analysis of QPLs, in particular, functional quantum languages. To this aim, we choose the Hybrid system as the tool in which to build our framework. Hybrid is a logical framework that supports higher-order abstract syntax, on top of which we develop an intuitionistic linear specification logic used for reasoning about QPLs. We provide a formal proof of some important meta-theoretic properties of this logic, and in addition, showcase a number of examples that can be tackled under the proposed framework.

Keywords: Quantum Lambda Calculus, Linear Logic, Hybrid, Coq

1 Introduction

Quantum computing has the potential to radically change the way computing is done. The existence of large-scale quantum machines would increase computational power exponentially and provide unbreakable security systems [4]. Quantum programming languages (QPLs) are an integral element required for achieving successful quantum machines, as they introduce quantum concepts at a high-level, allowing better understanding of quantum aspects and increasing access to research in quantum domains. QPLs were initiated by Knill [11] who provided a number of conventions to express quantum algorithms (i.e., quantum pseudo-code). The development of QPLs has evolved for both imperative languages, e.g., [16] and [19], and functional languages, e.g., [24] and [18]. QPLs are based on the QRAM computation model: the quantum state or data is stored in a RAM and the program is a sequence of primitive quantum operations (or controls) that update such a global quantum

[1] Email: myousri@uottawa.ca
[2] Email: afelty@uottawa.ca

https://doi.org/10.1016/j.entcs.2018.10.012

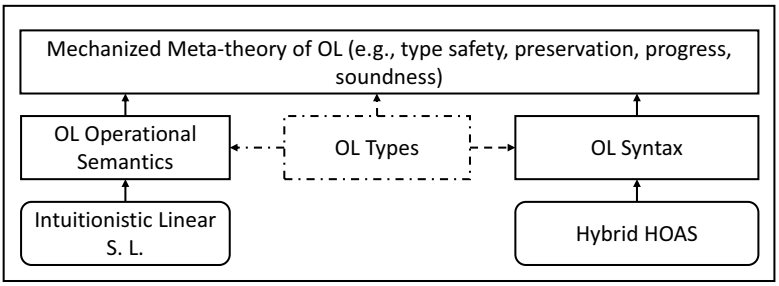

Fig. 1. Formal meta-level analysis framework for QPLs

state [11]. There is a trade-off between offering a high-level quantum language and capturing the full range of quantum aspects, e.g., no-cloning and superposition properties. Accordingly, it is crucial, when developing a QPL, to ensure that the proposed language, the computational model, and the operational semantics (as well as the type system if any) is practical, in terms of how the language respects the quantum properties and how it deals with *quantum non-determinism* (i.e., a quantum storage unit can hold both the zero and one values at the same time) and the *measurement process* (i.e., determining the exact value of a storage unit based on some probabilities). In this paper, we introduce a formal framework that aims to standardize meta-level analysis of quantum languages. Such a framework can help a language designer to focus on enhancing the language specification itself, leaving many of the details of correctness checking to the proposed framework. As a result, it becomes easier to make changes to the language as it evolves and study the effect of these changes, updating only the parts of proofs that are affected at each step.

Our concern in this paper is with functional quantum programming languages which typically map to quantum lambda calculi [20]. The major difference that a quantum calculus provides with respect to ordinary lambda calculus is that it addresses resource limitations: a quantum variable (i.e., quantum bit) is not duplicable (i.e., no cloning) and should be consumed only once and cannot be erased. This makes linear logic a good candidate for modeling the operational semantics for both typed and untyped quantum lambda calculi. To avoid involving the user in low-level details of language formalization, e.g., variable binding and substitution, we opt to use the Hybrid system [9], a two-level *logical framework* that supports higher-order abstract syntax (HOAS), implemented in both the Coq and Isabelle/HOL proof assistants. Figure 1 illustrates the design of our proposed framework for meta-level analysis of QPLs. OL stands for *object language*, which is the programming language subject to analysis. OL syntax contains the encoding of all possible expressions of a language (which often includes expressions that are not correct with respect to a well-formedness or other kind of judgment such as typing or evaluation). The use of Hybrid involves defining a *specification logic* (SL). An SL is developed independently from any OL, but is customizable through a parameter for atomic predicates used to express OL judgments, e.g., typing and evaluation rules. An SL is generally the formalization of a sequent calculus. We choose to implement intuitionistic linear logic which is well-suited to modeling the QRAM computation model that allows both intuitionistic resources (i.e., controls) and linear resources

(i.e., quantum data).

The middle block defines the syntax and basic properties of the types supported by an OL. Of course, this block is not included in the case of untyped quantum lambda calculi. QPL designers often choose to define untyped calculi to capture the maximum amount of quantum features (i.e., to build a quantum Turing complete language), which could be sacrificed by adding certain type systems. Nevertheless, the proposed framework supports both Turing complete and incomplete QPLs. The formalized syntax and semantics are then used to reason about the OL, e.g., proving subject reduction (type soundness) or Turing completeness.

We are implementing the proposed framework in Coq [13]. This paper presents completed work that builds on earlier (unpublished) work [12] where we formalize the Proto-Quipper language [18] along with some of its meta-theory. Here, we generalize the ideas from that case study to develop a general framework, focusing on two main ideas: generalizing the SL and providing a more complete and general set of meta-level properties that can be reused by many OLs, and illustrating its use on two different QPLs, namely Proto-Quipper and Q [24]. Although the work on Q is in its early stages, it illustrates the general nature of the framework. The rest of the paper discusses each box of Figure 1 in more detail.

2 Encoding OL Syntax in Hybrid

In this paper, we use the version of Hybrid that is implemented as a Coq library. The purpose of the first file in this library (Hybrid.v) is to provide support for expressing the syntax of OLs. At the core is a type `expr` that encodes a de Bruijn representation of lambda terms. It is defined inductively in Coq as follows:

```
Inductive expr: Set :=
  | CON: con -> expr
  | VAR: var -> expr
  | BND: bnd -> expr
  | APP: expr -> expr -> expr
  | ABS: expr -> expr.
```

Here, `VAR` and `BND` represent bound and free variables, respectively, and `var` and `bnd` are defined to be the natural numbers. The type `con` is a parameter to be filled in when defining the constants used to represent an OL. The library then includes a series of definitions used to define the operator `lambda` of type (`expr` → `expr`) → `expr`, which provides the capability to express OL syntax using HOAS. Expanding its definition fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user when reasoning about meta-theory. In fact, the user only needs `CON`, `VAR`, `APP`, and `lambda` to define operators for OL syntax. Two other predicates from the Hybrid library will appear in the proof development, `proper` : `expr` → `Prop` and `abstr` : (`expr` → `expr`) → `Prop` (where `Prop` is the type of Coq propositions). The `proper` predicate rules out terms that have occurrences of bound variables that do not have a corresponding binder (*dangling indices*). The `abstr` predicate is applied to arguments of `lambda` and rules out functions of type

(`expr` → `expr`) that do not encode object-level syntax.

We now give two examples filling in the middle right box in Figure 1.

Example 1. The following is a segment of the context-free grammar of Proto-Quipper [18], a typed QPL. We will use this example to explain general concepts about quantum lambda calculus:

$$
\begin{aligned}
T &::= \textbf{qubit} \mid T_1 \otimes T_2 \\
A &::= \textbf{qubit} \mid \textbf{bool} \mid A_1 \otimes A_2 \mid A_1 \multimap A_2 \mid !\,A \mid \mathrm{Circ}(T_1, T_2) \\
t &::= q \mid * \mid \langle t_1, t_2 \rangle \\
a &::= x \mid q \mid (t, C, a) \mid \texttt{True} \mid \texttt{False} \mid \langle a_1, a_2 \rangle \mid a_1 a_2 \mid \lambda x.a \mid \\
&\quad\ \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \mid \\
&\quad\ \texttt{let } \langle x, y \rangle = a_1 \texttt{ in } a_2
\end{aligned}
$$

The above grammar consists of two parts, the language types and expressions. We have two sorts of types, the quantum types T, and the general type A which allows typing of complex constructs. Type T is a subset of type A. The bang operator ! is used to create duplicable types from linear types. The arrow type $\multimap$ is used for functions (i.e., lambda abstraction), $\otimes$ for typing the tensor product of two expressions, and $\mathrm{Circ}(T_1, T2)$ for typing a circuit expression (t, C, a). This expression reads as follows: the circuit C has an input t of type T_1 and produces an output a of type T_2. Note that the input and the output types are quantum types.

Similar to types, Proto-Quipper has two sorts of expression: the pure quantum expressions t that include quantum bits and the tensor product of such qubits, and the general expressions a. The quantum circuit construct is considered a non-pure quantum expression since it allows for the output a to have a general expression sort. Nevertheless, this expression has to evaluate to an expression of quantum type. This condition is enforced by means of typing and reduction rules. Note that if we enforce the output expression to be of the pure quantum sort, then this language will not allow circuits to do any kind of computation, and thus the circuit construct loses its meaning and its main expressive power.

The lambda abstraction is the key expression of this language, and provides a way to distinguish between regular and quantum lambda calculi. For an expression $\lambda x.a$, the bound variable x typically appears linearly (i.e., only one instance) in a. In Proto-Quipper, the linearity of the bound variable is controlled using the type system, i.e., if the domain of the bound variable is of linear type (i.e., not preceded by bang operator) then x occurs once in a; if it is duplicable (i.e., preceded by bang) then x appears zero or more times in a.

Another important expression that is unique to the quantum lambda calculus is the *let-statement*. To highlight the difference with the usual *let-statement* in functional languages, we note that $\texttt{let } \langle x, y \rangle = a_1 \texttt{ in } a_2$ is not equivalent to:

$$\texttt{let } x = \texttt{fst } a_1 \texttt{ in let } y = \texttt{snd } a_1 \texttt{ in } a_2.$$

The main objective of the *let-statement* here is to allow for extracting the variables x and y at the same time with one access to the expression a_1. The reason for this is to respect the linearity constraint on the expression a_1; note that using `fst` and `snd` requires multiple accesses to a_1. Proto-Quipper's typing system ensures the valid construction of the *let-statement* in this language by enforcing a_1 to have linear tensor type. This way of destructing an expression whose type is a tensor product appears in quantum lambda calculi in general, but may take different styles, as we will see in the next example.

One more way in which this language differs from the regular lambda calculus is the inclusion of quantum variables q. $\mathcal{Q}$ is a countably infinite set of all possible quantum variables. These variables are linear by definition and their type is implicitly known (i.e., qubit), and thus it is not specified explicitly in Proto-Quipper programs.

For this object language, the type `con` is instantiated with the type `Econ`, implemented as:

```
Inductive Econ: Set :=
  Qvar: nat -> Econ | qPROD: Econ | qAPP: Econ | qABS: Econ |
  qIF: Econ | qLET: Econ | Crcons: nat -> Econ.
```

This definition provides a constant for each possible term in Proto-Quipper (i.e., in the language defined by the grammar a given above). Natural numbers are used to identify quantum variables and circuits since they are both isomorphic to the set of natural numbers. We do not represent term variables explicitly since they appear as bound variables in the HOAS representation of OL terms. Using these constants, we then define the OL expressions. For example, function application is defined as follows:

```
Definition App: qexp -> qexp -> qexp :=
  fun e1:qexp => fun e2:qexp => (APP (APP (CON qAPP) e1) e2).
```

The type `qexp` is the type `expr` discussed earlier with the parameter `con` instantiated with `Econ` for this OL. To understand the above definition, it may help to view the Hybrid `APP` constructor as an expressions concatenator, in this case, forming a list containing `e1` and `e2`. The first element of this "list" is `(CON qAPP)`, which provides an "annotation" indicating the kind of expression. This pattern is also seen in the formal definition of the tensor product, which is a binary operator like application. The definition only differs in the annotation, in this case `(CON qPROD)`:

```
Definition Prod: qexp -> qexp -> qexp :=
  fun e1:qexp => fun e2:qexp => (APP (APP (CON qPROD) e1) e2).
```

The next example is lambda abstraction where we can see clearly the use of HOAS:

```
Definition Fun : (qexp -> qexp) -> qexp :=
  fun f:qexp -> qexp => (APP (CON qABS) (lambda (fun x => f x))).
```

This example shows how HOAS deals with bound variables using functions of the

meta-level logic (which is Coq here), where `f` above has type `qexp -> qexp` and the bound variable `x` in the Coq function `fun x => f x` represents the bound variable in the Proto-Quipper lambda expression. Note the use of the `lambda` operator. As mentioned earlier, expanding its definition fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user, and does not appear in further proof development. There is another restriction needed in order for this expression to be a valid representation of a Proto-Quipper lambda abstraction. We must rule out Coq functions of type (`qexp` $\rightarrow$ `qexp`) that do not encode Proto-Quipper expressions. This check will be enforced using the `abstr` predicate when defining the prog clauses expressing OL semantics.

The general question of "valid" representation (also known as the *representational adequacy*) of the syntax of OLs is important and must be addressed for each OL considered. In particular, we must show that there is a bijection between the terms of the OL as specified by its grammar and the representation in a formal system (in this case Hybrid as implemented in Coq). Representational adequacy for the lambda calculus as an OL in Hybrid is discussed in [1] and proved in detail in [7]. Issues specific to adequacy proofs when using Hybrid are discussed in [9] and [10]. Proto-Quipper contains the lambda calculus as a sub-language, and representational adequacy for the full language is a straightforward extension of these other results. *Adequacy* of the representation of inference rules for OL judgments is also important. Such proofs are mainly carried out as pencil-and-paper proofs, because they involve both informal and formalized representations of syntax. For Hybrid, an important part of such proofs include a formal result, which we call *internal adequacy* theorems (see [10]). An example will be discussed in the next section.

The following definition is the formalization of the *let-statement* which is more complicated than the previous examples:

```
Definition Let:  (qexp -> qexp-> qexp) -> qexp -> qexp :=
  fun f:qexp -> qexp -> qexp => fun e1:qexp =>
  (APP (CON qLET)
    (APP (lambda (fun x => (lambda (fun y => f x y)))) e1)).
```

The `lambda` operator in Hybrid only supports lambda abstraction with a single bound variable. It does not directly support binding on a product of two variables, which is the case in the *let-statement*. One solution is to consider the product as a single bound variable, however, this requires the use of the `fst` and `snd` operations, which we want to avoid because, as discussed above their use violates the linearity constraint. Instead, we simply use cascading occurrences of `lambda` (two of them in this case) in a curried fashion where the expression is unfolded using two functions (i.e., two lambda abstractions). This representation respects the linearity condition.

Example 2. The following is an example of a segment of the context-free grammar of Q [24], an untyped QPL:

$$a \ ::= \ x \mid r \mid \langle a_1, a_2, ..., a_n \rangle \mid \lambda!x.a \mid \lambda x.a \mid \lambda\langle x_1, x_2, ..., x_n \rangle.a$$

where r refers to quantum variables, $\langle a_1, a_2, ..., a_n\rangle$ is a *tensor product* of n Q expressions (or a linear pattern). A major difference between Q and Proto-Quipper is that Q allows three types of lambda abstraction: over linear variables where the bound variable appears in the function body only once, over duplicable variables where bound variables may appear zero or more times, and over the tensor pattern of n elements. The distinction between the first two kinds is made in Proto-Quipper using its type system, but the third type is not supported in Proto-Quipper. However, we believe that it is sufficient to encode function abstraction over a tensor pattern using the same encoding technique that was used to represent the *let-statement* of Proto-Quipper, where $\lambda\langle x_1, x_2, ..., x_n\rangle.a$ is expanded to:

$$\lambda y_0.\texttt{let}\ \langle x_1, y_1\rangle = y_0\ \texttt{in}\ \texttt{let}\ \langle x_2, y_2\rangle = y_1\ \texttt{in}\ \cdots\ \texttt{let}\ \langle x_{n-1}, x_n\rangle = y_{n-2}\ \texttt{in}\ a$$

This conversion guarantees that $x_1, \ldots, x_n$ and $y_0, \ldots, y_{n-2}$ appear linearly in the body of the expression as long as $x_1, \ldots, x_n$ appear linearly in the original expression. This representation assumes that a two-operand *let* will be added to the Q representation. The adequacy of such a conversion must be proved to ensure that the expressions are equivalent and the language's expressiveness does not change.

In contrast to Proto-Quipper, the definition of `con` for Q will include two distinct constants for the two abstractions, namely `LABS` and `IABS`. It also has constants `qPROD` and `Qvar` as before. The third type of abstraction does not have a dedicated constant as we decided to encode it using the let-statement (i.e., `qLET`). Linear and intuitionistic abstraction are defined using an HOAS representation as follows:

```
Definition LAbs: (qexp -> qexp) -> qexp :=
  fun f:qexp -> qexp => (APP (CON LABS) (lambda (fun x => f x))).
Definition IAbs: (qexp -> qexp) -> qexp :=
  fun f:qexp -> qexp => (APP (CON IABS) (lambda (fun x => f x))).
```

Note that at the syntax level both lambda abstractions are the same except for the constant annotation. The real difference will be imposed in the encoding of the OL semantic rules as we will see in the next section. Note also the difference between the definition of `Fun` from Proto-Quipper which contains `(lambda f)` and the two definitions here, which contain `(lambda (fun x => f x))`. The two terms are η-equivalent; in general, terms in Hybrid are equivalent up to η-conversion, which means that either form can be used to encode such OL expressions.

3 Encoding the SL and OL Inference Rules

The sequents of the intuitionistic linear logic we adopt as an SL (bottom left of Figure 1) have the form $\Gamma; \Delta \vdash_\Pi G$, where G is a formula, Γ is an intuitionistic context of formulas (a set of formulas), Δ is a linear context (a multiset of formulas), and Γ and Δ contain only atomic formulas. The restriction to atomic formulas is sufficient for the examples we have considered so far, and simplifies reasoning using Hybrid. In [3], an intuitionistic SL is defined that allows more general formulas in contexts. If this kind of generality is needed in future case studies in our framework,

$$\frac{}{\Gamma; A \vdash A}\ \text{l\_init} \qquad \frac{}{\Gamma, A; . \vdash A}\ \text{i\_init} \qquad \frac{}{\Gamma; \Delta \vdash \top}\ \top\text{-R}$$

$$\frac{\Gamma; \Delta_1 \vdash B \qquad \Gamma; \Delta_2 \vdash C}{\Gamma; \Delta_1, \Delta_2 \vdash B \otimes C}\ \otimes\text{-R} \qquad \frac{\Gamma; \Delta \vdash B \qquad \Gamma; \Delta \vdash C}{\Gamma; \Delta \vdash B \& C}\ \&\text{-R}$$

$$\frac{\Gamma, A; \Delta \vdash B}{\Gamma; \Delta \vdash A \Rightarrow B}\ \Rightarrow\text{-R} \qquad \frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B}\ \multimap\text{-R} \qquad \frac{\Gamma; \Delta \vdash B[y/x]}{\Gamma; \Delta \vdash \forall x.B}\ \forall\text{-R}$$

$$\frac{\Gamma, A, A; \Delta \vdash B}{\Gamma, A; \Delta \vdash B}\ \text{Contraction} \qquad \frac{\Gamma, A_1; \Delta \vdash B}{\Gamma, A_1, A_2; \Delta \vdash B}\ \text{Weakening}$$

$$\frac{\begin{array}{l} A \longleftarrow [G_1, \ldots, G_m][G'_1, \ldots, G'_n] \in [\Pi] \\ \Gamma; . \vdash G_i \quad (i = 1, \ldots, m) \\ \Gamma; \Delta_i \vdash G'_i \quad (i = 1, \ldots, n) \end{array}}{\Gamma; \Delta_1 \ldots \Delta_n \vdash A}\ bc$$

Fig. 2. $\mathcal{MALL}$ sequent rules

it should be straightforward to extend our linear SL using the same approach as in that paper. Π is a set of formulas (also called *clauses*) expressing the inference rules of an OL, which we omit when presenting the sequent rules because it is fixed for each OL and does not change within a proof. They can be considered as a fixed subset of Γ. In particular, we are interested in the $\mathcal{MALL}$ fragment of intutionistic linear logic, which includes the connectives $\otimes$ for multiplicative conjunction, & for additive conjunction, $\multimap$ for linear implication, $\Rightarrow$ for intuitionistic implication, and $\top$ for the universal resource consumer.

The sequent rules of this logic are presented in Figure 2 (where $\cdot$ represents an empty context). Note that the ";" is used to separate the linear context from the intuitionistic one whereas the "," is used to append a hypothesis to a context. We now discuss the rules from top to bottom. There are two initial rules, the linear rule (l_init) strictly prohibits the existence of any hypothesis inside Δ except A, and does not care about the contents of Γ. The intuitionistic rule (i_init) strictly requires an empty linear context, whereas A should be in the intuitionistic context. We can use $\otimes$ if its operands can be proven linearly at the same time, i.e., they do not share linear resources. On the other hand, additive conjunction is used when the operands are sharing the linear resources. Because they both consume all resources, only one of them can be made available at a time. The implication rules ($\Rightarrow$-R and $\multimap$-R) vary based on which context the antecedent A comes from. The $\forall$-R rule has the usual proviso that y does not appear in Γ, Δ, or B. The usual rules for contraction and weakening apply only to the intuitionistic context.

In the *bc* rule, $[\Pi]$ represents all possible instances of clauses in Π (clauses with instantiations for all variables quantified at the outermost level). Clauses are not

arbitrary formulas, but have the restricted form shown by the first premise of this rule. Applying this rule in a backward direction corresponds to *backchaining* on a clause in Π, instantiating the universal quantifiers so that the head of the clause matches A. There is one hypothesis for each *subgoal*, both linear and intutionistic.

There are a number of formalizations of linear logic, e.g., [5] in Abella and [15,23] in Coq. The main purpose of these formalizations is handling the meta-analysis of different fragments of linear logic. In contrast, we use linear logic as an intermediate logic in which to study object languages such as QPLs. Our objective is broader since it includes both. In particular, we prove meta-level theorems about linear logic and gear our choice of theorems toward those that are useful in analyzing any QPL. Our formalization is inspired by the work in [9], which implements ordered intuitionistic linear logic as a specification logic in the Isabelle/HOL version of Hybrid, where it is used to study a continuation-machine presentation of the operational semantics of a functional language, which is much simpler than the QPLs considered here.

In our implementation, formulas of the SL are defined as an inductive type `oo`. This definition introduces constants for each connectives of the SL:

```
Inductive oo: Set :=
  | atom: atm -> oo
  | T: oo
  | Conj: oo -> oo -> oo
  | And: oo -> oo -> oo
  | Imp: atm -> oo -> oo
  | lImp: atm -> oo -> oo
  | All: (expr con -> oo) -> oo.
```

where the `atom` constructor accepts an atomic formula of type `atm` and casts it into an SL formula. The type `atm` is defined for each object language and typically includes predicates such as `typeof` and `is_exp`, where the former is a binary predicate relating a QPL expression and its type and the latter is a unary well-formedness predicate. The constructor `T` corresponds to the universal consumer, `Conj` corresponds to multiplicative conjunction, and `And` to additive conjunction. The type constructors `Imp` and `lImp` corresponds to intuitionistic and linear implication, respectively, where in both cases, the formula on the left must be an atom. The `All` constructor takes a function as an argument, and thus the bound variable in the quantified formula is encoded using lambda abstraction in Coq.

The SL sequent rules are then formalized as an inductive predicate `seq` of type `list atm -> list atm -> oo -> Prop`. The first list of atoms refers to the intuitionistic context, whereas the second list contains linear atoms. Below are some examples of sequent rules formalized in the `seq` definition:

```
Inductive seq: list atm -> list atm -> oo -> Prop :=
  | l_init: forall (A:atm) (IL:list atm), seq IL [A] (atom A)
  | i_init: forall (A:atm) (IL:list atm),
      In A IL -> seq IL nil (atom A)
```

```
| s_tt: forall (IL LL:list atm), seq IL LL T
| m_and: forall (B C:oo) (IL L L1 L2:list atm),
    split L L1 L2 -> seq IL L1 B -> seq IL L2 C ->
    seq IL L (Conj B C)
| a_and: forall (B C:oo) (IL L: list atm),
    seq IL LL B -> seq i IL L C -> seq  IL L (And B C)
| s_bc: forall (A: atm) (IL LL: list atm) (lL iL: list oo) (b: oo),
    prog A iL lL ->
    splitseq L [] iL -> splitseq  IL LL lL ->
    seq IL LL (atom A)
```

In this set of rules, we do not include structural rules for intuitionistic contexts, such as weakening and contraction. They will be proved admissible later as part of the meta-theorems about this sequent calculus that will be important to include to help users reason about OLs.

Notice that we use the type `list` to model logical contexts. Using a list to model sets, where duplication does not matter and order has no meaning, is convenient for representing the intuitionistic context. In particular, the Coq type `list` can behave as sets without the need to impose restrictions on them. However, this does not work directly for the linear context where duplication does matter, though order does not (i.e., as in multisets). For this case, we must add a restriction to make lists behave implicitly as multisets. This restriction appears in the multiplicative conjunction rule where we connect the linear contexts for the sequents with conclusions `B` and `C` using the `split` predicate. In particular, in `split L L1 L2`, list `L` is "split" into the two lists `L1` and `L2`. This predicate can be viewed as a multiset union operator since it preserves the number of occurrences of each element inside `L1` and `L2` without necessarily maintaining the order of appearance in `L`. For example, the split of `[A;B;C]` can be `[A;B]` and `[C]`, `[C]` and `[A;B]`, `[A]` and `[C;B]`, or `[B]` and `[C;A]`. Later, we will state a meta-level structural theorem expressing that this predicate works as required, and does not affect provability. Alternatively, we could have used the existing `multiset` library in Coq. However, we found this library too weak for our purposes, and not rich enough in comparison to the `list` library. Since it was not our focus to build a library for multisets, we chose the "implicit" definition using lists.

Another important thing we want to highlight: in the `s_bc` rule, the predicate `splitseq` is used to check the provability of a list of subgoals. The predicate `splitseq` is used twice; one for the intuitionistic subgoals `iL` under the empty linear context, the other one for the linear subgoals `lL`. The predicates `seq` and `splitseq` are defined using Coq's mutual induction. One can look at `splitseq` as a generalization of `seq`, where a list of goals are proved instead of just one. The `splitseq` predicate is defined with the help of the `split` predicate, which ensures that the union of the linear contexts used to prove the subgoals respect the multiset behavior.

To keep things simple here, we omit an additional natural number argument to the `seq` predicate that allows proofs by induction over the height of a sequent

derivation. The reader is referred to [13] for full details.

The backchaing rule also depends on the inductive predicate `prog` of type `atm -> list oo -> list oo -> Prop`, which encodes the inference rules of an OL (such as its operational semantics). The formula `(prog A IL L)` reads as follows: an atom `A` (the conclusion of an inference rule of the OL) is true in an OL if we can prove each member of the list of intuitionistic subgoals `IL` and of the linear subgoals `LL` (together representing the premises of an OL rule). Examples will be given later for our two QPLs.

The implementation of the SL is significantly generalized from that in [12] in the sense that we have proved more meta-theoretic properties. Since these properties can be applied to any OL, this allows for a more complete validation of OL properties. For example, in addition to proving that weakening and contraction are admissible, we have proved the admissibility of cut rules for both intuitionistic and linear contexts. The intutionistic rule is formalized as follows:

```
Lemma seq_cut_aux: forall (a:atm) (b:oo) (IL L:list atm),
  seq IL L b -> seq (remove eq_dec a IL) [] (atom a) ->
  seq (remove eq_dec a IL) L b.
```

The theorem states that if we remove all instances of the hypothesis `a` from the list of intuitionistic hypotheses `IL`, and `a` is found to be provable under the new list of hypotheses, then eliminating `a` does not affect the provability of `b`. Note that the `remove` (a Coq list operation) requires a proof of the decidability of equality at type `atm` as an argument.

On the other hand, the linear version of the cut elimination rule allows the removal of only one instance of the linear resource `a`:

```
Lemma seq_cut_linear: forall (a:atm) (b:oo) (IL L L':list atm),
  seq IL L b -> seq IL L' (atom a) ->
  seq IL (L' ++ remove_one eq_dec a L) b).
```

We have defined the `remove_one` function (it is not available in the Coq list library) to remove just one copy of a given formula from a list. Note that `++` is the Coq notation for the list append operator.

The admissibility of weakening over the intuitionistic context is stated as follows:

```
Theorem seq_weakening_cor: forall (b:oo) (il il' ll:list atm),
  seq il ll b ->
  (forall a :atm, In a il -> In a il') ->
  seq il' ll b.
```

It important to highlight that our version of linear logic is not ordered, i.e., the order in which the resources appear inside the linear context does not affect provabilty. This is a significant difference with respect to the linear logic SL defined in [9]. To insure that the formalized logic does respect this condition, we have proved the exchange property of the linear context (i.e., the linear context behaves as a multiset), stated as follows:

```
Theorem seq_exchange_cor: forall (b:oo) (il ll' ll:list atm),
```

```
seq il ll b ->
(forall a, count_occ eq_dec ll a = count_occ eq_dec ll' a) ->
seq il ll' b.
```

where the `count_occ` operation counts the number of occurrences of an element in a given list. For two linear contexts `ll` and `ll'` that have same number of occurrences of each atom, the order they appear in the list does not matter; they are exchangeable in any valid proof in the formalized SL. The proof of this theorem involves numerous lemmas related to the definitions of `count_occ` and `remove_one`, i.e., related to the multiset behavior of the linear context.

The following two examples illustrate the middle left box in Figure 1.

Example 3. Continuing Example 1, the Proto-Quipper typing judgment has the form $\Gamma; Q \vdash a : A$. In this sequent, Γ is a finite set of typing declarations of the form $x : A$ where x is a variable and A is a type (A may have the form $!C$ or not). Q is a quantum context containing a finite set of quantum variables, typically the free quantum variables in a. The following are two examples (out of four) of the Proto-Quipper lambda abstraction typing rules:

$$\frac{\Gamma, x : A; Q \vdash b : B}{\Gamma; Q \vdash \lambda x.b : A \multimap B}\ \lambda_1 \qquad \frac{\Delta, x : !A; Q \vdash b : B}{\Delta; . \vdash \lambda x.b : !A \multimap B}\ \lambda_2$$

In rule λ_1, for a Proto-Quipper expression b whose types is B and linearly dependent on variable x (i.e., x appears only once in b) whose linear type is A, the lambda abstraction over x then yields a function of type $(A \multimap B)$. This rule is then encoded as part of the `prog` clauses dedicated to the Proto-Quipper language as follows:

```
| lambda1l: forall (T1 T2:qtp) (E:qexp -> qexp),
  abstr E -> validT (bang T1) -> validT T2 ->
  prog (typeof (Fun (fun x => E x)) (arrow T1 T2)) []
   [(All (fun x:qexp => Imp (is_qexp x)
     (lImp (typeof x T1) (atom_ (typeof (E x) T2)))))]
```

Here, the type `qtp` encodes Proto-Quipper types (whose definition was omitted from Example 1), and `validT` checks for certain well-formedness conditions of elements of this type, such as ruling out two consecutive occurrences of `bang`. The `typeof` predicate is a constructor of the inductively defined type `atm`, which associates an expression with its type, and `is_qexp` is a constructor of `atm` used to annotate well-formed expressions. As mentioned earlier, the `abstr` predicate rules out functions of type `qexp -> qexp` that do not encode OL syntax. In this clause, `lImp` is used to ensure that the bound variable `x` is linear in the body of the function `E`, i.e., this assumption will go into the linear context. Note also the use of `Imp`; well-formedness assumptions are always part of the intuitionistic context in our formalization. Finally, note that the subgoal occurs in the list of linear subgoals. This is because the type of the whole expression `Fun (fun x => E x)` is linear; it may also contain quantum variables, which can each only occur once. The list of intuitionistic subgoals is empty.

$$\frac{[C, a] \to [C', a']}{[C, if\ a\ then\ b\ else\ c] \to [C', if\ a'\ then\ b\ else\ c]}\ cond$$

$$\frac{}{[C, if\ True\ then\ b\ else\ c] \to [C, b]}\ ifT$$

$$\frac{}{[C, if\ False\ then\ b\ else\ c] \to [C, c]}\ ifF$$

Fig. 3. The *if-statement* reduction rules

Note that the type `oo` defined earlier along with all its constructors, such as `atom`, encode a general version of formulas that must be instantiated for each OL. The constant `atom_` appearing in the `prog` clauses such as the one above is the instantiated version for Proto-Quipper.

The following `prog` clause encodes the λ_2 rule, i.e., the non-linear bound variable case:

```
| lambda1i: forall (T1 T2:qtp) (E:qexp -> qexp),
  abstr E -> validT (bang T1) -> validT T2 ->
  prog (typeof (Fun (fun x => E x)) (arrow (bang T1) T2)) []
   [(All (fun x:qexp => Imp (is_qexp x)
     (Imp (typeof x (bang T1)) (atom_ (typeof (E x) T2)))))]
```

Note the use of `Imp` (the second occurrence in the above clause) because the type of the bound variable is duplicable. The subgoals, however, must still be linear as the function body can still depend on quantum variables. In contrast, these conditions are required to be in the list of intutionistic subgoals when the function body is non-linear, i.e., it does not contain linearly typed variables, in particular, quantum variables.

Figure 3 shows an example of the reduction rules of Proto-Quipper, in particular, for the *if-statement*. The term $[C, a]$ is a circuit closure, where a is a Proto-Quipper expression that depends on the quantum variables produced by the circuit C, i.e., the outputs of the circuit. The corresponding formal presentation of these rules are as follows:

```
| ifr: forall C C' b b' a1 a2,
  valid_c C (If b a1 a2) -> valid_c C' (If b' a1 a2) ->
  ~(is_value b) ->
  prog (reduct C (If b a1 a2) C' (If b' a1 a2))
    [atom_ (reduct C b C' b');
     atom_ (is_qexp a1); atom_ (is_qexp a2)] []
| truer: forall C a1 a2,
  valid_c C (If (CON TRUE) a1 a2) ->
  prog (reduct C (If (CON TRUE) a1 a2) C a1)
    [atom_ (is_qexp a1); atom_ (is_qexp a2)] []
| falser: forall C a1 a2, valid_c C (If (CON FALSE) a1 a2) ->
```

```
prog (reduct C (If (CON FALSE) a1 a2) C a1)
  [atom_ (is_qexp a1); atom _(is_qexp a2)] []
```

where `reduct` is a constructor of `atm` that encodes $\rightarrow$, and thus associates an expression with its reduced expression, and `valid_c` ensures that a circuit closure `[C,a]` forms a valid closure, i.e., the quantum variables of `a` belong to the output of circuit `C`. Note that the rule `ifr` is only applicable if `b` is not a value; otherwise this rule could be applied an infinite number of times without achieving any progress. One of the other two rules is applied when `b` is a value, i.e., it represents the boolean `True` or `False`.

Returning to the issue of adequacy, internal adequacy theorems mentioned earlier generally state that if a particular OL judgment is provable, then all of the terms in this judgment are well-formed. In particular, the well-formedness judgment for an OL, such as `is_qexp` for Proto-Quipper, must only hold for terms of type `qexp` that represent OL terms. We omit the details, since we have not described the rules for `is_qexp` here, and instead describe it informally. Using this judgment, we can prove internal adequacy for the `typeof` judgment, for example. This particular theorem states that if a sequent of the form `seq iL lL (atom_ (typeof M T))` is provable then the sequent `seq iL [] (atom_ (is_qexp M))` is also provable, under an assumption about the form of the contexts `iL` and `lL`. In other words, if `iL` and `lL` are well-formed contexts and `M` can be proven to inhabit a type `T`, then `M` is a well-formed Proto-Quipper expression in context `iL`. (See [10] for examples of internal adequacy for simpler OLs.)

Example 4. We choose to illustrate the Q language by considering well-formedness of terms of the form $\lambda x.a$ and $\lambda !x.a$ from Example 2. Let Γ and Δ be contexts of intuitionistic and linear term variables, respectively. The term $\lambda x.a$ (respectively $\lambda !x.a$) is well-formed in $\Gamma; \Delta$ if a is well-formed in $\Gamma, x; \Delta$ (respectively $\Gamma; \Delta, x$). The `prog` clauses for these lambda expressions are as follows:

```
| lambda1: forall (M:qexp -> qexp), abstr M ->
  prog (is_qexp (LAbs M)) []
    [All (fun x:qexp => lImp (is_qexp x) (atom_ (is_qexp (M x))))]
| lambda2: forall (M:qexp -> qexp), abstr M ->
  prog (is_qexp (IAbs M)) []
    [All (fun x:qexp => Imp (is_qexp x) (atom_ (is_qexp (M x))))]
```

Note the difference between the two rules: linear implication is used to define the linear lambda abstraction (which enforces the requirement that the function body contain one copy of `x`), whereas intuitionistic implication is used to define intuitionistic lambda abstraction, which allows multiple copies of `x` or even zero occurrences.

Now, we conclude our formalization by showing an example meta-theorem, namely subject reduction (or type soundness). The following statement represents an abbreviated form of this theorem for the Proto-Quipper language, using the definitions provided earlier:

```
Theorem subject_reduction: forall IL LL C C' a a',
  seq IL [] (atom_ (reduct C a C' a')) ->
```

```
seq IL LL (atom_ (typeof a A)) ->
seq IL LL (atom_ (typeof a' A)).
```

It is abbreviated because certain conditions on the contexts `IL` and `LL` that are required for adequacy are omitted. This theorem states that if an expression `a` produced by a quantum circuit `C` reduces to an expression `a'` produced by a quantum circuit `C'`, then the reduced expression maintains the same type as the original expression. For an untyped programming language, the corresponding statement of the above theorem will be slightly different, where the `typeof` atoms will be replaced by `is_qexp` atoms. Progress theorems are another type of property that is important for QPLs to satisfy. Proving such a property for Proto-Quipper is part of our immediate future work. This theorem along with subject reduction will result in a formalization of the principal results in [18].

4 Conclusion

We have proposed a meta-level analysis framework for functional QPLs implemented using the Hybrid system with a linear specification logic. The framework provides a standard way to tackle common components and concepts of QPLs, e.g., operational semantics and type safety. Formalization examples of the Proto-Quipper and Q languages have been presented to show the practical potential of the proposed system.

Future work includes carrying out a more complete formalization of these languages in addition to others, e.g., [8,17,21]. Formalizing multiple examples will help to improve our framework. In particular, it will involve building infrastructure that decreases the amount of work required for an average user to handle a complete formalization. The QWIRE language in [17], for instance, is implemented in Coq and is expressive enough to include languages like Proto-Quipper. The work in that paper focuses on proving properties of quantum programs and program transformations, such as proving that a program meets its formal specification. Our framework will provide a platform in which to also study the meta-theory of this language. Another direction of future work is to expand our framework to provide support for reasoning about particular programs and operations on them.

Our plans for future work also include generalizing the SL. The logic programming approach we use was first introduced in the $FO\lambda^{\Delta N}$ logic [14], and is also used in other Hybrid SLs, e.g., [9,10]. An alternative is to use a more general *focusing* style as introduced in [2], and used to implement e.g., the formalization of a generalized intuitionistic SL in Hybrid [3] as well as the classical linear logic in [23]. The former was inspired by the an earlier formalization of the same logic in Abella [22]. The latter adopts the parametric approach to HOAS [6]. It would be interesting to implement the same linear logic in Hybrid, which uses a more direct style of HOAS, and then compare the two.

References

[1] Ambler, S., R. L. Crole and A. Momigliano, *Combining higher order abstract syntax with tactical theorem proving and (co)induction*, in: *15th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*, Lecture Notes in Computer Science (2002), pp. 13–30.

[2] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), pp. 297–347.

[3] Battell, C. and A. Felty, *The logic of hereditary Harrop formulas as a specification logic for Hybrid*, in: *11th Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)* (2016), pp. 3:1–3:10.

[4] Brassard, G., C. Crepeau, R. Jozsa and D. Langlois, *A quantum bit commitment scheme provably unbreakable by both parties*, in: *34th Annual Symposium on Foundations of Computer Science*, 1993, pp. 362–371.

[5] Chaudhuri, K., L. Lima and G. Reis, *Formalized meta-theory of sequent calculi for substructural logics*, in: *Postproceedings of the 11th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2016)*, Electronic Proceedings in Theoretical Computer Science **332** (2017), pp. 57–73.

[6] Chlipala, A., *Parametric higher-order abstract syntax for mechanized semantics*, in: *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008, pp. 143–156.

[7] Crole, R. L., *The representational adequacy of Hybrid*, Mathematical Structures in Computer Science **21** (2011), pp. 585–646.

[8] Díaz-Caro, A. and G. Dowek, *Simply typed lambda-calculus modulo type isomorphisms*, CoRR **arXiv/1501.06125** (2014).

[9] Felty, A. P. and A. Momigliano, *Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax*, Journal of Automated Reasoning **48** (2012), pp. 43–105.

[10] Felty, A. P., A. Momigliano and B. Pientka, *The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2—a survey*, Journal of Automated Reasoning **55** (2015), pp. 307–372.

[11] Knill, E., *Conventions for quantum pseudocode*, Technical Report LAUR-96-2724, Los Alamos National Laboratory (1996), https://www.osti.gov/scitech/servlets/purl/366453.

[12] Mahmoud, M. Y. and A. P. Felty, *Formalization of metatheory of the Quipper quantum programming language in a linear logic* (2017), http://www.site.uottawa.ca/~afelty/dist/HybridProtoQuipper17.pdf.

[13] Mahmoud, M. Y. and A. P. Felty, *Quantum programming language Coq scripts* (2017), https://bitbucket.org/snippets/myousri/Gj7qX.

[14] McDowell, R. and D. Miller, *Reasoning with higher-order abstract syntax in a logical framework*, ACM Transactions on Computational Logic **3** (2002), pp. 80–136.

[15] Olivier Laurent, *YALLA: an LL library for Coq* (2017), https://perso.ens-lyon.fr/olivier.laurent/yalla/.

[16] Ömer, B., "A Procedural Formalism for Quantum Computing," Master's thesis, Technical University of Vienna (1998).

[17] Rand, R., J. Paykin and S. Zdancewic, *QWIRE practice: Formal verification of quantum circuits in Coq*, in: *Postproceedings of the 14th International Conference on Quantum Physics and Logic (QPL 2017)*, Electronic Proceedings in Theoretical Computer Science **266**, 2018, pp. 119–132.

[18] Ross, N. J., "Algebraic and Logical Methods in Quantum Computation," Ph.D. thesis, Dalhousie University (2015), CoRR arXiv:1510.02198 [quant-ph].

[19] Sanders, J. W. and P. Zuliani, *Quantum programming*, in: *5th International Conference on Mathematics of Program Construction (MPC)*, Lecture Notes in Computer Science (2000), pp. 80–99.

[20] Selinger, P. and B. Valiron, *A lambda calculus for quantum computation with classial control*, Mathematical Structures in Computer Science **16** (2006), pp. 527–552.

[21] Vizzotto, J. K., B. Calegaro and E. K. Piveta, *A double effect λ-calculus for quantum computation*, in: *17th Brazilian Symposium on Programming Languages (SBLP)*, Lecture Notes in Computer Science (2013), pp. 61–74.

[22] Wang, Y., K. Chaudhuri, A. Gacek and G. Nadathur, *Reasoning about higher-order relational specifications*, in: *15th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)* (2013), pp. 157–168.

[23] Xavier, B., C. Olarte, G. Reis and V. Nigam, *Mechanizing linear logic in Coq*, in: *Postproceedings of the 12th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2017)*, Electronic Proceedings in Theoretical Computer Science, 2018.

[24] Zorzi, M., *On quantum lambda calculi: a foundational perspective*, Mathematical Structures in Computer Science **26** (2016), pp. 1107–1195.