Noname manuscript No.
(will be inserted by the editor)

Formalization of Metatheory of the Quipper Quantum Programming Language in a Linear Logic

Mohamed Yousri Mahmoud · Amy P. Felty

July 2018

Abstract We develop a linear logical framework within the Hybrid system and use it to reason about the type system of a quantum lambda calculus. In particular, we consider a practical version of the calculus called Proto-Quipper, which contains the core of Quipper. Quipper is a new quantum programming language under active development and recently has gained much popularity among the quantum computing communities. Hybrid is a system that is designed to support the use of higher-order abstract syntax (HOAS) for representing and reasoning about formal systems implemented in the Coq Proof Assistant. In this work, we extend the system with a linear specification logic (SL) in order to reason about the linear type system of Quipper. To this end, we formalize the semantics of Proto-Quipper by encoding the typing and evaluation rules in the SL, and prove type soundness.

Keywords Proto-Quipper · Quantum Programming Languages · Linear Logic · Hybrid · Higher-Order Abstract Syntax · Coq

1 Introduction

Quipper is a functional programming language designed for implementing quantum algorithms [10]. The mathematical foundations of the Proto-Quipper fragment, which retains much of the important expressive power of the full language, is developed in [27]. As the authors themselves have noted [28], there is a great deal of subtlety in the definitions of the syntax and semantics, and many details were fine-tuned during the process of proving the type soundness result. This process certainly would have benefited from formalization and the ability to recheck proofs after each change in the definitions. Quipper is a relatively new language, and additional metatheory will be proved as it

School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada
E-mail: myousri@uottawa.ca, afelty@uottawa.ca

is developed. Providing an environment within a proof assistant that allows the developers to simultaneously develop and formalize this metatheory is a central goal of our work.

The Hybrid system [7] provides support for reasoning about *object languages* (OLs) such as programming languages and other formal systems using *higher-order abstract syntax* (HOAS), sometimes referred to as "lambda-tree syntax" [22,23]. It is implemented as a two-level system, an approach first introduced in the $FO\lambda^{\Delta IN}$ logic [18]. Using this approach, the specification of the semantics of the OL and the meta-level reasoning about it are done within a single system but at different levels. In the case of Hybrid, an intermediate level is introduced by inductively defining a *specification logic* (SL) in Coq, and OL judgments are encoded in the SL. The version of Hybrid we use here is implemented as two distinct libraries in the Coq Proof Assistant. The first we adopt without change, while the development of a new version of the second is part of the contributions of this work.

The first Hybrid library provides support for expressing the syntax of OLs. Using HOAS, binding constructs of the OL are encoded using lambda abstraction in Coq. For instance, for Proto-Quipper, type `qexp` will represent terms or programs, and `App` of type `qexp` $\rightarrow$ `qexp` $\rightarrow$ `qexp` and `Fun` of type (`qexp` $\rightarrow$ `qexp`) $\rightarrow$ `qexp` will represent application and abstraction, respectively, of the linear lambda calculus, which forms the core of the Proto-Quipper language. Lambda abstraction in Proto-Quipper is a *binder* because the name of a variable is bound in the body of the abstraction. For example, the term $\lambda x.\lambda y.xy$ can be encoded as (`Fun` (λx.`Fun` (λy.(`App` x y)))). Note that `qexp` cannot be defined inductively because of the (underlined) negative occurrence of `qexp` in the type of `Fun`. Hybrid provides an underlying low-level de Bruijn representation of terms to which the HOAS representation can be mapped internally; the user works directly with the higher level HOAS representation.

Using such a representation, α-conversion at the meta-level directly represents bound variable renaming at the object-level, and meta-level β-conversion can be used to directly implement object-level substitution. As a consequence, we avoid the need to develop large libraries of lemmas devoted to operations dealing with variables, such as capture-avoiding substitution, renaming, and fresh name generation. In proof developments that use a first-order syntax, such libraries are often significantly larger than the formalization of the main metatheory results.

The second Hybrid library defines the SL and provides support for encoding OL judgments and inference rules, and for reasoning about them. The need for different levels arises in Hybrid because there are OL judgments that cannot be encoded as inductive propositions in Coq. As an example, we consider assigning simple types to lambda terms. The standard rule is as follows:

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \rightarrow T'}$$

Let `qtp` be the type of OL types, and let `arr` be a constant of type `qtp` $\rightarrow$ `qtp` $\rightarrow$ `qtp` for constructing arrow types. Let `typeof` be a predicate expressing

the relation between a term and its type. If we consider the HOAS encoding of the above rule, using for example, the techniques introduced in the logical framework LF [11], we can encode the above rule as the following formula:

$$\begin{array}{l}\forall T, T' : \texttt{qtp}.\ \forall t : \texttt{qexp} \to \texttt{qexp}. \\ \quad (\forall x : \texttt{qexp}.\ \texttt{typeof}\ x\ T \to \texttt{typeof}\ (t\ x)\ T') \to \\ \quad\quad \texttt{typeof}\ (\texttt{Fun}\ t)\ (\underline{\texttt{arr}\ T\ T'})\end{array}$$

The second line contains a formula with embedded implication and universal quantification (known as *hypothetical* and *parametric* judgments), in particular universal quantification over variable x of type `qexp` and an internal assumption (underlined) about the type of this x. We note that the `typeof` predicate cannot be expressed inductively because this underlined occurrence is negative. A two-level system solves this problem by encoding OL predicates inside the SL, where negative occurrences are allowed. We will see how this is done in Section 5.1, where we encode rules such as the typing rule for lambda abstraction in the quantum lambda calculus.

The linear SL we implement here is an extension of the ordered linear SL implemented in [7] and the ordered and linear SLs presented in [17]. We extend and adapt this previous work to the much larger case study considered here, and we design the new SL to be general so that it can be adopted directly for reasoning about a variety of other OLs with linear features.

We formalize the key property of *type soundness* (also called *subject reduction*) of Proto-Quipper, which requires several important lemmas about context subtyping.

The issue of the *adequacy* of HOAS representations is important (see [11]), which here means that we must prove that the encoding in Hybrid does indeed represent the intended language. The work described here includes extending previous work on adequacy to our setting, where both the OL (Proto-Quipper) and the SL (a linear logic) are more complex than those considered previously.

We begin in the next section with background material on Proto-Quipper. Then in Section 3, we present the encoding of types and of the subtyping relation in Coq. Proto-Quipper types and subtyping can be encoded directly using inductive types. There are no binders and no substitution. One of the strengths of Hybrid as implemented in Coq is that it is straightforward to combine such direct encodings of OL syntax with encodings that use the HOAS and SL facilities provided by Hybrid.

In Section 4, we present the Hybrid system including both background information on the first Coq library, as well as our new implementation of the second library, which encodes our linear SL in Coq and develops some important general meta-theoretic properties that help in reasoning about OLs.

The encoding of Proto-Quipper terms and of the typing rules, along with some properties about them appear in Section 5. The encoding of reduction rules is presented in Section 6, along with the statement of type soundness and a discussion of its formal proof.

We discuss adequacy of our encoding in Section 7, and finally, we conclude and discuss related and future work in Section 8.

The files of our formalization are publicly available [15].

2 Proto-Quipper

We give a brief background of the Proto-Quipper language, focusing on the aspects that are required for understanding the formalization in later sections. Proto-Quipper is based on the quantum lambda calculus and focuses on Quipper's abilities to generate and manipulate quantum circuits [27]. The types and terms of Proto-Quipper are defined by the following grammars:

$$
\begin{array}{rcl}
T, U & ::= & \mathbf{qubit} \mid 1 \mid T \otimes U \\
A, B & ::= & \mathbf{qubit} \mid 1 \mid !1 \mid \mathbf{bool} \mid !\,\mathbf{bool} \mid A \otimes B \mid !(A \otimes B) \mid \\
 & & A \multimap B \mid !(A \multimap B) \mid \mathrm{Circ}(T, U) \mid !(\mathrm{Circ}(T, U)) \\
t & ::= & q \mid * \mid \langle t_1, t_2 \rangle \\
a, b, c & ::= & x \mid q \mid (t, C, a) \mid \texttt{True} \mid \texttt{False} \mid \langle a, b \rangle \mid * \mid ab \mid \lambda x.a \mid \\
 & & \mathit{rev} \mid \mathit{unbox} \mid \mathit{box}^T \mid \texttt{if } a \texttt{ then } b \texttt{ else } c \mid \\
 & & \texttt{let } * = a \texttt{ in } b \mid \texttt{let } \langle x, y \rangle = a \texttt{ in } b
\end{array}
$$

Proto-Quipper distinguishes between *quantum data types* (T, U) and *types* (A, B) where the former is a subset of the latter, and similary between *quantum data terms* (t) and *terms* (a, b, c). Here x and y are *term variables* from a set $\mathcal{V}$, q is a *quantum variable* from a set $\mathcal{Q}$, and C is a *circuit constant* from a set $\mathcal{C}$. The sets $\mathcal{V}$, $\mathcal{Q}$, and $\mathcal{C}$ are all assumed to be countably infinite.

Most types and term constructs come directly from the quantum lambda calculus, e.g., [29]. The type $\mathrm{Circ}(T_1, T_2)$ represents the set of all circuits having an input interface of type T_1 and an output interface of type T_2. A circuit constant C represents a low-level quantum circuit, and a term (t, C, a) represents a circuit as Proto-Quipper data, where t is a structure representing a finite set of inputs to C, and similarly a represents a finite set of outputs. In Proto-Quipper, it is assumed that two functions exists, In and Out, from $\mathcal{C}$ to the set of all subsets of $\mathcal{Q}$, where $In(C)$ and $Out(C)$ are a superset of the set of input and output quantum variables, respectively, of circuit C. The terms *rev*, *unbox*, and box^T represent functions on circuits. We refer the reader to [27] for further description of these types and terms.

In quantum computing, variable cloning is prohibited. This features is reflected in Proto-Quipper by using the modal operator !, where variables having type $!A$ are called duplicable and can be cloned whereas variables of types that do not follow this pattern are called linear and cannot be cloned or copied. Note that instead of introducing a general ! operator on types, we restrict it to prohibit more than one consecutive occurrence of the bang operator !. This presentation of types differs from the one in [27].[1]

[1] In general, when we stray from the original presentation, our intention is to simplify formalization, and we only do so when there is a clear equivalence to the original. In this case, we simplify the formalization of the subtyping relation without changing the semantics of types. As we will discuss in the next section, making this kind of change also led to the discovery of a small mistake in the original presentation.

$$\frac{}{qubit <: qubit} \qquad \frac{}{1 <: 1} \qquad \frac{}{bool <: bool}$$

$$\frac{A_1 <: B_1 \qquad A_2 <: B_2}{A_1 \otimes A_2 <: B_1 \otimes B_2} \qquad \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \multimap B_1 <: A_2 \multimap B_2}$$

$$\frac{T_2 <: T_1 \qquad U_1 <: U_2}{Circ(T_1, U_1) <: Circ(T_2, U_2)} \qquad \frac{A <: B}{!\,A <: B}\ \dagger \qquad \frac{A <: B}{!\,A <: !\,B}\ \dagger$$

† Note that the last two rules have the proviso that A and B do not have a leading !.

Fig. 1 Subtyping rules for Proto-Quipper

The use of the bang operator ! introduces a subtyping relation among types: A variable of type $!A$ obviously is also of type A. The subtyping rules of Proto-Quipper are shown in Figure 1. The subtyping relation $<:$ is the smallest relation on types satisfying these rules.

The Proto-Quipper typing judgment has the form $\Phi; Q \vdash a : A$. In this sequent, Φ 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). In the presentation of the rules, Φ always appears in two parts $\Phi', !\Psi$ where the types in the latter all follow the pattern $!\,A$, while those in the former never contain a leading !. Q is a quantum context containing a finite set of quantum variables, typically the free quantum variables in a. Also a is a term and A is a type. The typing rules are shown in Figure 2. We write $\cdot$ to represent an empty context. The ";" is used to separate the typing context from the quantum context whereas the "," is used to append two contexts. The rules containing $!^n$ abbreviate two distinct rules, one where $n = 0$, i.e., there is no leading !, and one where $n = 1$. These rules are taken from [27], with some modifications that do not change the semantics. For example, when $!^n$ appears in the rules there, n can be any natural number. The restriction here takes into account our modifcation to the syntax of types discussed above. Also, the ax_x and ax_c rules replace a single rule for subtyping in [27]. Here, we give an initial rule for term variables (ax_x), similar to the one for quantum variables (ax_q). When using LF-style hypothetical and parametric judgments, such rules for variables are implicit; they do not appear in the encoding of the inference rules in the SL. In general, we avoid explicit treatment of variables whenever possible to get the full advantage of HOAS encodings. In addition, the subtyping rule in [27] is restricted only to term variables. Again to avoid specific reasoning about variables, our ax_c rule is valid for any valid expression of Proto-Quipper.

In the (cst) rule c ranges over the set $\{box, unbox, rev\}$, and we write $box(T, U)$ as $box^T(U)$ to make explicit that the constant box is always annotated with its type (see the grammar).

A_{box^T}, A_{unbox}, and A_{rev} are defined as follows:

$$\begin{aligned} A_{box^T}(U) &:= !(T \multimap U) \multimap !\,\mathrm{Circ}(T, U) \\ A_{unbox}(T, U) &:= \mathrm{Circ}(T, U) \multimap !(T \multimap U) \\ A_{rev}(T, U) &:= \mathrm{Circ}(T, U) \multimap !\,\mathrm{Circ}(U, T) \end{aligned}$$

$$\frac{}{!\Psi; q \vdash q : \mathbf{qubit}}\ ax_q \qquad \frac{}{!\Psi, x : A; \cdot \vdash x : A}\ ax_x$$

$$\frac{!\Psi; \cdot \vdash a : !^n A \qquad !^n A <: B}{!\Psi; \cdot \vdash a : B}\ ax_c \qquad \frac{!\, A_c(T,U) <: B}{!\Psi; \cdot \vdash c : B}\ cst$$

$$\frac{}{!\Psi; \cdot \vdash * : !^n 1}\ *_i \qquad \frac{}{!\Psi; \cdot \vdash \texttt{True} : !^n \mathit{bool}}\ \top \qquad \frac{}{!\Psi; \cdot \vdash \texttt{False} : !^n \mathit{bool}}\ \bot$$

$$\frac{\Phi, x : !^n A; Q \vdash b : B}{\Phi; Q \vdash \lambda x.b : !^n A \multimap B}\ \lambda_1 \qquad \frac{!\Psi, x : !^n A; \cdot \vdash b : B}{!\Psi; \cdot \vdash \lambda x.b : !(!^n A \multimap B)}\ \lambda_2$$

$$\frac{\Phi_1, !\Psi; Q_1 \vdash c : A \multimap B \qquad \Phi_2, !\Psi; Q_2 \vdash a : A}{\Phi_1, \Phi_2, !\Psi; Q_1, Q_2 \vdash ca : B}\ app$$

$$\frac{\Phi_1, !\Psi; Q_1 \vdash a : !^n A \qquad \Phi_2, !\Psi; Q_2 \vdash b : !^n B}{\Phi_1, \Phi_2, !\Psi; Q_1, Q_2 \vdash \langle a, b\rangle : !^n (A \otimes B)}\ \otimes_i$$

$$\frac{\Phi_1, !\Psi; Q_1 \vdash b : !^n (B_1 \otimes B_2) \qquad \Phi_2, !\Psi, x : !^n B_1, y : !^n B_2; Q_2 \vdash a : A}{\Phi_1, \Phi_2, !\Psi; Q_1, Q_2 \vdash \texttt{let}\ \langle x, y\rangle = b\ \texttt{in}\ a : A)}\ \otimes_e$$

$$\frac{\Phi_1, !\Psi; Q_1 \vdash b : !^n 1 \qquad \Phi_2, !\Psi; Q_2 \vdash a : A}{\Phi_1, \Phi_2, !\Psi; Q_1, Q_2 \vdash \texttt{let}\ * = b\ \texttt{in}\ a : A}\ *_e$$

$$\frac{\Phi_1, !\Psi; Q_1 \vdash b : \mathit{bool} \qquad \Phi_2, !\Psi; Q_2 \vdash a_1 : A \qquad \Phi_2, !\Psi; Q_2 \vdash a_2 : A}{\Phi_1, \Phi_2, !\Psi; Q_1, Q_2 \vdash \texttt{if}\ b\ \texttt{then}\ a_1\ \texttt{else}\ a_2 : A}\ if$$

$$\frac{Q_1 \vdash t : T \qquad !\Psi; Q_2 \vdash a : U \qquad In(C) = Q_1 \qquad Out(C) = Q_2}{!\Psi; \cdot \vdash (t, C, a) : !^n\, \texttt{Circ}(T,U)}\ circ$$

Fig. 2 Typing rules for Proto-Quipper

3 Proto-Quipper Types

In the previous section, the types supported in Proto-Quipper were presented using a context free grammar, where there are two classes of types. Here, we consider a single "universal" class of types, and then define predicates that discriminate between the two classes. We start the formalization of the Proto-Quipper types (file ProtoQuipperTypes.v) by defining an inductive type of the universal class:

```
Inductive qtp: Type :=
  qubit: qtp | one: qtp | bool: qtp
| tensor: qtp -> qtp -> qtp | arrow: qtp -> qtp -> qtp
| circ: qtp -> qtp -> qtp | bang: qtp -> qtp.
```

Certainly, the above definition does not model Proto-Quipper types. For instance, it does allow for the parameters of the circuit type constructor to be of the general class, whereas the argument expressions are supposed to be of the quantum data type. Accordingly, we define the inductive predicate `valid` that captures the notion of quantum data types (grammar T, U):

```
Inductive valid: qtp -> Prop :=
  Qubit: valid qubit
```

```
| One: valid one
| Tensor: forall A1 A2, valid A1 -> valid A2 ->
  valid (tensor A1 A2).
```

Then, we define the general `validT` predicate to identify the general types (grammar A, B):

```
Inductive validT: qtp -> Prop :=
  vQubit: validT qubit
| bQubit: validT (bang qubit)
⋮
| vTensor: forall A B, validT A -> validT B ->
  validT (tensor A B)
| bTensor: forall A B, validT A -> validT B ->
  validT (bang (tensor A B))
⋮
| vCirc: forall A B, valid A -> valid B ->
  validT (circ A B)
| bCirc: forall A B, valid A -> valid B ->
  validT (bang (circ A B)).
```

In longer definitions such as this one, we often omit parts when it is clear how to fill in the complete idea of the definition. We have proved that `(valid A)` implies `(validT A)`, confirming that class T, U is a subclass of A, B.

The last step in the formalization of the Proto-Quipper types is the development of the subtyping relation. The following inductive proposition directly encodes the rules of Figure 1.

```
Inductive Subtyping: qtp -> qtp -> Prop :=
  QubitSub: Subtyping qubit qubit
| OneSub: Subtyping one one
⋮
| CircSub: forall A1 A2 B1 B2,
  Subtyping A2 A1 -> Subtyping B1 B2 ->
  validT (circ A1 B1) -> validT (circ A2 B2)->
  Subtyping (circ A1 B1) (circ A2 B2)
| BangSub1: forall A B, Subtyping A B ->
  validT (bang A) -> Subtyping (bang A) B
| BangSub2: forall A B, Subtyping A B ->
  validT (bang A) -> Subtyping (bang A) (bang B).
```

The main difference between this definition and the original subtyping rules in [27] are the last two rules. As mentioned in Section 2, they are stated so that they prevent consecutive applications of the ! operator. The first of these rules (`BangSub1`) is concerned with the weakening of subtype `A` by adding the bang operator. The second rule concerns weakening both sides of the subtyping relation. The use of the predicate `validT` throughout the definition ensures

that non-Quipper types are ruled out. This presentation of the subtyping rules for the bang operator make the formal proof of the transitivity of the subtyping relation much easier; in particular it avoids a lot of unneeded induction cases. An important theorem, which states that the subtyping relation implies the validity of its arguments, is stated below.

```
Theorem SubAreVal: forall A B, Subtyping A B ->
  validT A /\ validT B.
```

Given the above definition of Proto-Quipper subtyping, we successfully verified all the required specifications reported in [27], which ensures that the implemented definition follows the intended behavior. In the following, we list examples of the formally proven properties:

```
Theorem Subtyping_Prop1: forall B,
  Subtyping qubit B -> B = qubit.
```

Similar theorems have been proven for the other base types `one` and `bool`. The following specification ensures that whenever the top-level type constructor of the super-type is the bang operator, then the subtype should be too.

```
Theorem Subtyping_Prop6: forall A B1,
  Subtyping A (bang B1) ->
  exists A1, A = (bang A1) /\ Subtyping A1 B1.
```

The second conjunct of this theorem can be concluded by inversion using the `Bang_Sub2` rule. It is important to know that this property cannot be proven for the original subtyping relation reported in [27]. Let us consider the case of `(Subtyping !one !!one)`, from which one cannot conclude that `(Subtyping one !one)`. Thanks to the formal proof, we were able to spot this ill-formed condition. The author has been contacted and confirmed the mistake.

Similar to the bang constructor, we prove that if the outermost type constructor of the subtype (not the super-type) is `arrow` then the super-type should be too:

```
Theorem Subtyping_Prop2: forall A1 A2 B,
  Subtyping (arrow A1 A2) B ->
  exists B1 B2,
  B = arrow B1 B2 /\ Subtyping B1 A1 /\ Subtyping A2 B2.
```

Similar theorems have been developed for the other type constructors `tensor` and `circ`. Finally, we provide two important properties of the subtyping relation: reflexivity and transitivity:

```
Theorem sub_ref: forall A, validT A -> Subtyping A A.
```

Note that reflexivity is subject to the validity of `A`, i.e., it belongs to the Proto-Quipper types.

```
Theorem sub_trans: forall A B C,
  Subtyping A B -> Subtyping B C -> Subtyping A C.
```

Note that transitivity does not require that validity of `A`, `B`, and `C` since it is implicitly imposed from the subtyping antecedents (see `Theorem SubAreVal`).

This concludes the Proto-Quipper types formalization, where we considered the formal development of valid Proto-Quipper types and the subtyping relation on them.

4 Two-Level Hybrid

As mentioned, the purpose of the first Hybrid library (implemented as Hybrid.v) is to provide support for expressing the higher-order abstract syntax of OLs. This file is described in Section 4.1, focusing on the aspects that are required for understanding the formalization in later sections. The second library (implemented as LSL.v) is described in Section 4.2 and contains the encoding of the specification logic (SL), which provides the two-level reasoning capabilities as described earlier. Hybrid allows the use of different SLs, and as mentioned, one of the contributions of this paper is to present a new one. In fact, our goal is to provide a general framework for reasoning about a large class of OLs that have linear features, and preliminary work toward this goal is discussed in [16]. This aspect of our work follows the tradition of a variety of linear logical frameworks that have been introduced in the literature (though none that we know of is maintained in a such a way that we could easily adopt it for our work). In addition, there is growing body of OLs with linear features that would benefit from such a framework. Examples of such frameworks and applications are discussed in Section 8.

4.1 Expressing Syntax of Object Languages in Hybrid

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, including negative occurrences in the types of binders. Expanding its definition fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user when reasoning about metatheory. 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`. 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, discussed further in Section 7.

As mentioned, the type `con` is actually a parameter in the Hybrid library. This will become explicit when presenting Coq definitions, where we write (`expr con`) as the type used to express OL terms. The constants for Proto-Quipper will be introduced later as an inductive type called `Econ` and the type `qexp` mentioned earlier is an abbreviation for `(expr Econ)`.

Before describing our SL in the next section, we include here a summary of the steps of the implementation of any SL. Formulas of the SL are implemented as an inductive type `oo`. This definition introduces constants for the connectives of the SL and their Coq types. The rules of the SL are defined as a Coq inductive proposition, where each clause represents one rule. This definition is called `seq` and has one argument for each of the elements of a sequent, which includes the context(s) of assumptions and the conclusion of the sequent. It may also include a natural number used to keep track of the height of a deriviation.

The rules of the OL are also defined as an inductive propostion called `prog`. In our case, `prog` defines the inference rules for well-formedness of Proto-Quipper terms, typing of Proto-Quipper terms, and the reduction relation for evaluating Proto-Quipper terms. Its definition uses the capability to express hypothetical and parametric judgments. In the library file defining the SL, `prog` is a parameter to the definition of `seq`.

When encoding an OL, the file Hybrid.v must be imported since it is (usually) required to represent the syntax of the OL, while the file containing the SL must be imported when defining `prog`. When an element of the syntax can be defined directly as an inductive type, however, there is no need to import any Hybrid files. This is the case for the syntax of types of Proto-Quipper as we have seen. The terms of Proto-Quipper, however, cannot be defined inductively; instead we will define a HOAS representation, which uses the `lambda` operator and other constructors of type `expr`. As mentioned, using Hybrid provides the flexibility to mix both kinds of representations.

4.2 A Linear Specification Logic

In this section, we will give a brief account of linear logic, highlighting differences with minimal intuitionistic logic, which is the main SL used in Hybrid historically. Then we present the sequent calculus of our selected version of linear logic, namely an intuitionistic linear logic. It has both an intuitionistic

and a linear context of assumptions. The latter is important for modeling the type system of Proto-Quipper.

In minimal intuitionistic logic, there are three logical connectives ($\wedge$,$\vee$ and $\Rightarrow$), in addition to the logical constants True and False. A sequent in this logic has the form $\Gamma \vdash C$ where Γ is a logical context (a set of formulas) and C is a formula. If a sequent $\Gamma \vdash C$ is valid in intuitionistic logic, then the sequent obtained by adding a hypothesis B to the context, i.e., $\Gamma, B \vdash C$, is also valid. This is called context weakening. If a sequent of the form $\Gamma, B, B \vdash C$ is valid in intuitionistic logic, then the sequent $\Gamma, B \vdash C$ is also valid. This is called context contraction. These two key structural properties of intuitionistic logic are primarily prohibited in linear logic, where we deal with the context as a collection of resources, i.e., the hypotheses are considered as resources that can be used one time and they must be consumed (or used). Accordingly, $\Gamma \vdash C$ does not guarantee the linear validity of $\Gamma, B \vdash C$, and $\Gamma, B, B \vdash C$ does not guarantee the linear validity of $\Gamma, B \vdash C$. In addition, linear logic has two types of logical connectives: the multiplicative connectives ($\otimes$, $\parr$ and $\multimap$), and the additive connectives &, $\oplus$ and $\Rightarrow$. In addition to the intuitionistic constants, there is the universal consumer constant $\top$, which can consume any linear resource (i.e., hypothesis). Our choice for SL is a slightly different version of the standard linear logic, namely intuitionistic multiplicative linear logic, with the two kinds of contexts mentioned above: the intuitionistic context Γ (a set of formulas) and the linear context Δ (a multiset of formulas). The contraction and weakening rules are permitted for the intuitionistic context Γ. The logic has two classes of formulas—goals and program clauses—whose syntax is:

$$\begin{array}{lll} \text{Goals} & G & ::= A \mid \top \mid \forall x : \tau\ G \mid A \Rightarrow G \mid G_1 \& G_2 \mid G_1 \otimes G_2 \mid A \multimap G \\ \text{Clauses} & P & ::= \forall (A \longleftarrow [G_1, \ldots, G_m][G'_1, \ldots, G'_n]) \end{array}$$

In the above grammar, A is an atomic formula. Note that this logic includes universal quantification over typed variables. Here, τ is a primitive type. It will be instantiated with `qexp` in our case study. A clause in the form above consists of two *lists*, the first one of intuitionistic goals, the other of linear ones. It is an abbreviation for the formula:

$$\forall (G_1 \Rightarrow \cdots \Rightarrow G_m \Rightarrow G'_1 \multimap \cdots \multimap G'_n \multimap A)$$

where the outer $\forall$ represents quantification over all free variables in $G_1, \ldots, G_m$, $G'_1, \ldots G'_n, A$, whose types may be τ or $\tau \to \tau$, where τ is primitive. Thus, our logic is second-order in the sense that it is possible to quantify over functions (of type `qexp -> qexp` in our case).

Sequents of this logic have the form $\Gamma; \Delta \vdash_\Pi G$, where G is a goal formula, Γ is an intuitionistic context of formulas, Δ is a linear context, and Γ and Δ contain only atomic formulas. Π is a set of program clauses, which we omit when presenting the rules because it is fixed and does not change within a proof. This format emphasizes the view of this calculus as a non-deterministic logic programming interpreter, which is a feature of most SLs implemented in the two-level style. See e.g., [7] and [21]. The sequent rules of such a logic are

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

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

$$\frac{\Gamma, A; \Delta \vdash B}{\Gamma; \Delta \vdash A \Rightarrow B}\ \Rightarrow\text{-R} \qquad \frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B}\ \multimap\text{-R}$$

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

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

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

Fig. 3 Intuitionistic Linear Logic Sequent Rules

shown in Figure 3. There are two initialization rules. The linear rule (l_init) strictly prohibits the existence of any hypothesis inside Δ except A, and it does not care about the contents of Γ. The intuitionistic rule (i_init) strictly requires an empty Δ whereas A should be part of Γ. We can use & if its operands can be proven linearly at the same time, i.e., all the required linear resources are available in the contexts of both premises (&-R). Recall that a linear resource can be used only once. On the other hand, additive conjunction requires that the resources be split and used in the proof of only one premise ($\otimes$-R). This connective is suitable when the operands are sharing the linear resources. 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.

In the *bc* rule, $[\Pi]$ represents all possible instances of clauses in Π (clauses with instantiations for all variables quantified at the outermost level). 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 intuitionistic.

The first step towards the formalization of the linear specification logic in Coq is defining an inductive type `oo` for formulas given by the G and P grammars:

```
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 the formula type `oo`. The type `atm` is defined for each OL and typically includes the atomic relations or predicates of the OL, e.g., `typeof` for Proto-Quipper typing (see Section 5.2). 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 next step is defining the sequent rules themselves. This step is done using an inductive predicate definition as follows:

```
Inductive seq: nat -> list atm -> list atm -> oo -> Prop  :=
| s_bc: forall (i:nat) (A:atm) (IL LL:list atm)
  (lL iL:list oo), prog A iL lL ->
  splitseq i L [] iL -> splitseq i IL LL lL ->
  seq (i+1) IL LL (atom A)
⋮
| s_all: forall (i:nat) (B:expr con -> oo) (IL LL:list atm),
  (forall x:expr con, proper x -> seq i IL LL (B x)) ->
  seq (i+1) IL LL (All B)
with
splitseq: nat -> list atm -> list atm -> list oo -> Prop :=
| ss_init: forall (i:nat) (IL:list atm), splitseq i IL [] []
| ss_general: forall (i:nat) (IL lL1 lL2 lL3:list atm)
  (G:oo) (Gs:list oo),
  split lL1 lL2 lL3 -> seq i IL lL2 G ->
  splitseq i IL lL3 Gs -> splitseq i IL lL1 (G::Gs).
```

where we use the variables `IL` to represent the intuitionistic context, and `LL` for the linear one. The definition of `seq` follows the standard rules described earlier. Only a representative set of the inference rules of the formal definition is shown above. The natural number argument allows proofs by induction over the height of a sequent derivation, which we often use. This argument can be ignored when doing proofs by structural induction. The reader is referred to [15] for the full definition. 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; once for the intuitionistic subgoals `iL` under the empty linear context, and once for the linear subgoals `lL`. When we discuss the rules for `prog` later, we will say that we must *intuitionistically prove* the goals in `iL` and *linearly prove* the goals in `lL`. The predicates `seq` and `splitseq` are defined using Coq's mutual induction. The inductive definition of `splitseq` is at the very end of the definition. In the case when the list of goals is non-empty, the head subgoal `G` is proven under the linear context `lL2` and the remaining list of subgoals `Gs` is proven under the context `lL3` if and only if there exists a context `lL1` such that `split lL1 lL2 lL3`. We describe the `split` predicate

by example (and omit its definition here): 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]. The idea of the split is that the two sublists divide the big list, regardless of the order of elements inside the sublist. This is to ensure that there are no shared linear resources between `lL2` and `lL3`. Finally, `(prog A iL lL)` represents a formula from the program context defining the rules of the OL (i.e., Π). Later (in Section 5.2), we will see the implementation of `prog` for Proto-Quipper.

The rule for `All` has an argument of function type `expr con` $\rightarrow$ `oo`, unlike the other rules, since quantification is over terms of the OL. Recall that `con` is a parameter to the type `expr`, and must be implemented for each OL. Note that we restrict `x` to terms satisfying the `proper` predicate (terms without dangling indices).

The implemented specification logic has been validated by proving a number of essential properties. We show two here. The first property is a cut-elimination rule for the intuitionistic context:

```
Lemma seq_cut_aux:
  forall (i j:nat) (a:atm) (b:oo) (il ll:list atm),
  seq i il ll b -> In a il ->
  seq j (remove eq_dec a il) [] (atom a) ->
  seq (i+j) (remove eq_dec a il) ll 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` function takes a proof that equality at type `atm` is decidable. Since `atm` is a parameter to the SL, so is `eq_dec`. The second property is the weakening property for the intuitionistic context:

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

This concludes the formalization of the linear specification language and some of its metatheory. In the following sections, we will present Proto-Quipper as an OL that benefits from this logic, where will give a concrete implementation for the parameters presented in this section, in particular, `atm`, `con`, and `prog`, as well as prove decidability of equality for our instantiation of `atm`.

5 Encoding Proto-Quipper Programs and Semantics in Hybrid

In this section, we discuss a key aspect of this work, where we present the encoding of Proto-Quipper in the Hybrid framework. This includes the formalization of Proto-Quipper syntax as a concrete implementation of the types `con` and `atm`, and the main parts of the program context `prog`, which includes the typing rules in Figure 2. We then present a number of theorems that are important for proving type soundness of Proto-Quipper.

5.1 Encoding Proto-Quipper Terms

Recall that in Hybrid, `con` is a parameter to the type `expr`. The implementation of `con` for an OL typically includes all constants that appear in the language. This includes the names of the operations supported by the OL, e.g., `if` and `let`. The implementation of `con` for Proto-Quipper is as follows:

```
Inductive Econ: Set :=
| qABS: Econ | qAPP: Econ | qPROD: Econ
| qLET: Econ | sLET: Econ | qCIRC: Econ | qIF: Econ
| BOX: qtp -> Econ | UNBOX: Econ | REV: Econ
| TRUE: Econ | FALSE: Econ | STAR: Econ
| Qvar: nat -> Econ | Crcons: nat -> Econ.
```

This definition might cause some confusion since it does not differentiate between constants like `TRUE` and `FALSE`, and operations like lambda abstraction `qABS` and the let statement `qLET`. Actually, these are just constants and this definition does not include any semantics or functionality by itself. The functionality of these operations will be addressed next. In our formalization, we encode all quantum variables of Proto-Quipper as constants using the constant `Qvar`, which maps natural numbers to quantum variables. Similarly, the constant `Crcons` models the names of quantum circuits, i.e., the circuit constant C (see Section 2). The argument of type `qtp` to the `BOX` operator encodes the type superscript in box^T. As mentioned earlier, `qexp` is an abbreviation for `(expr Econ)`.

```
Definition qexp: Set := expr Econ.
```

As stated earlier, all OL constructs are encoded using `CON`, `VAR`, `APP` and `lambda`. These definitions are the only place where the constructors of the `expr` type and the `lambda` operator are seen explicitly. Once the new constants are defined, we use only these. We start with the simplest item in the language, namely variables:

```
Definition Var: var -> qexp := VAR Econ
```

The definition is simple as it just uses the Hybrid `VAR` parameterized with the Proto-Quipper constants `Econ`. Another simple example of encoding Proto-Quipper operations in Hybrid is function application:

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

To help the reader to understand the above definition, it is better to view the Hybrid constructor `APP` as a concatenation operator. Another possible definition is `(APP (CON qAPP) (APP e1 e2))`. One might wonder if it is useless to add the constant `qAPP`. Remember that the `APP` constructor is used to represent other program statements, e.g., the `if` statement. Therefore, we need to add such constants for each type of expression so we can identify statements with different meanings. Similarly, we formally define the product and circuit statements as follows:

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

Definition Circ (e1:qexp) (i:nat) (e2:qexp) : qexp :=
  APP (APP (APP (CON qCIRC) e1) (CON (Crcons i))) e2.
```

Now, let us turn to a Proto-Quipper construct that is a bit more difficult, namely function abstraction.

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

This expression has an argument that is a function of type `(qexp -> qexp)`. To better understand this construct, we give some examples. Let `f` be the function `fun x => App (Var 0) x`. Consider the term `(lambda f)`. If definitions are expanded, the Hybrid operator `lambda` disappears and the result is the de Bruijn format of this term which replaces the bound variable `x` by the correct de Bruijn index. For the above example, `(lambda f)` unfolds to `ABS (APP (VAR ECON 0) (BND 0))`. As stated, we never need to expand such definitions. Now that we have the full encoding of Proto-Quipper terms, we can return to the example term from the introduction, $\lambda x.\lambda y.xy$, which is encoded as `(Fun (fun x:qexp => (Fun (fun y:qexp => (App x y)))))`.

Similarly, we handle the more difficult case of the `Let` expression.

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

Recall that the `let` statement in Proto-Quipper is restricted to product expressions, i.e., it always takes the form `let` $\langle x, y\rangle = a$ `in` b. Therefore, we have two function abstractions, e.g., `fun x => fun y => App y x`. Since the `lambda` is only defined for functions of type `exp -> exp` (not `exp -> exp -> exp`), we have to make two applications of `lambda` in the way presented in the above definition in order to satisfy the typing condition of `lambda`. This is the first case study using Hybrid that requires a function of more than one argument to represent OL syntax.

We note that terms in Hybrid are equivalent up to η-conversion, so the body of the definition of `Fun` and `Let` operators, respectively, could also be written:

```
(APP (CON qABS) (lambda (fun x => (f x)))).
(APP (CON qLET)
  (APP (lambda (fun x => (lambda (fun y => (f x y))))) b)).
```

The formal encoding of the other Proto-Quipper expressions is similar to the definitions presented above [15].

We also define the following predicate that holds for expressions that only involve quantum variables, the star constant, and the product constructor `Prod`.

```
Inductive quantum_data: qexp -> Prop :=
  vQVAR: forall i, quantum_data (CON (Qvar i))
```

```
| vSTAR: quantum_data (CON STAR)
| vTENSOr: forall a b, quantum_data a -> quantum_data b ->
  quantum_data (Prod a b).
```

This subset of expressions are those whose types satisfy the `valid` predicate.

5.2 Encoding Proto-Quipper Semantics

The main purpose of this section is to discuss the implementation of the atomic predicates `atm` and the program context `prog` for Proto-Quipper. Here, `atm` contains three constructors, one that relates a Proto-Quipper type `qtp` to a Proto-Quipper expression `qexp`, one for representing reduction rules, and one that identifies valid (well-formed) expressions of Proto-Quipper.

```
Inductive atm : Set :=
| typeof : qexp -> qtp -> atm
| reduct : Econ -> qexp -> Econ -> qexp -> atm
| is_qexp : qexp -> atm.
```

It is important to clarify that the formalization presented in the previous section does not guarantee that all instances of the type `qexp` are valid in Proto-Quipper. (It just defines expressions that we are interested in.) This will be done as part of the program context `prog` with the help of the constructor `is_qexp`.

Now, we turn to the crucial step in our whole formalization, the implementation of `prog`. The formal definition of such a predicate is quite long. Accordingly, we choose to present it rule by rule, where we address only the major rules, and others can be found in [15]. We discuss clauses for `is_qexp` and `typeof` here, and leave the discussion of `reduct` to Section 6.

Now that we have instantiated the parameters `con` and `atm`, we can define the first two abbreviations below.

```
Definition oo_ := oo atm Econ.
Definition atom_: atm -> oo_ := atom Econ.
Definition seq_: nat -> list atm -> list atm -> oo_ -> Prop :=
  seq prog.
Definition splitseq_:
  nat -> list atm -> list atm -> list oo_ -> Prop :=
  splitseq prog.
```

The third and fourth are useful once we have completed the definition of `prog`. Thus the general form of sequents of the SL will be written `seq_ n IL LL G`, where the arguments are the height of the proof, the intuitionistic and linear contexts of assumptions of type `atm`, and the conclusion (formula on the right of the turnstile), respectively. We will sometimes omit the height argument for readability when it is not important for the discussion, and in particular in the statements of theorems, `seq_ IL LL G` will mean that there exists an `n` such that `seq_ n IL LL G`.

The `prog` predicate has type `atm -> list oo_ -> list oo_ -> Prop`, whose arguments are: the atomic statement, the list of intuitionistic subgoals, and the list of linear subgoals. Recall that it appears in the definition of `seq` in the `s_bc` clause, which implements the *bc* rule in Figure 3. It is the implementation of the program clauses Π. In the upcoming rules, this predicate reads as follows: the atomic statement is valid in the context of Proto-Quipper program if the list of intuitionistic and linear subgoals can be proven using the linear specification logic presented in Figure 3 (as implemented by `seq`). We start by presenting examples of syntax rules that define valid expressions inside Proto-Quipper:

```
| starq: prog (is_qexp (CON STAR)) [] []
| trueq: prog (is_qexp (CON TRUE)) [] []
| boxq: prog (is_qexp (CON BOX)) [] []
```

Constants of Proto-Quipper are unconditionally valid; the list of linear and intuitionistic subgoals are empty. Similar clauses are included in `prog` for the constants `FALSE`, `UNBOX`, and `REV`.

Note that we have defined the syntax of terms in Section 2 as grammars. As argued in [6], grammars contain implicit information, and specifying well-formedness as inference rules makes some of this information explicit, which is useful for formally defining valid terms. For example, consider the sublanguage of Proto-Quipper terms containing term variables, application, and function abstraction. One possible set of rules expressing valid syntax is as follows:

$$\frac{is\_qexp\ x \in \Phi}{\Phi \vdash is\_qexp\ x} \qquad \frac{\Phi \vdash is\_qexp\ a \qquad \Phi \vdash is\_qexp\ b}{\Phi \vdash is\_qexp\ ab}$$

$$\frac{\Phi, is\_qexp\ x \vdash is\_qexp\ a}{\Phi \vdash is\_qexp\ \lambda x.a}$$

Here, Φ is an explicit context of variables, and a term is only considered well-formed if all of its free variables come from this set. Stated formally in our setting, for a sequent `seq_ IL [] (atom_ (is_qexp a))` to be provable, for each free variable `x` of type `qexp` that appears in `a`, the atom `is_qexp x` must be in the intuitionistic context `IL`. Note that the list of linear subgoals is empty. This is the case for all well-formedness of syntax rules, as the context elements expressing validity of an expression can be used as many times as we want, i.e., they are infinitely consumable resources.

The following clause encodes the well-formedness of the application expression:

```
| apq: forall E1 E2:qexp,
  prog (is_qexp (App E1 E2))
   [And (atom_ (is_qexp E1)) (atom_ (is_qexp E2))] []
```

As mentioned, the list of linear subgoals is empty. In the intuitionistic goals, notice is the use of additive conjunction. Actually, we cannot use multiplicative conjunction to express an intuitionistic subgoal according to the specification

logic defined in Section 4.2. The use of `And` for intuitionistic subgoals has the same power as the use of multiplicative conjunction because the use of the intuitionistic context involves infinitely consumable resources, and hence we can show the validity of `E1` and `E2` from the same context. Similar rules are defined for the `Prod` and `Slet` expressions, and also the `If` expression, where we have three sub-expressions instead of two.

The following clause corresponds to the function abstraction (`Fun`) case.

```
| lambdaq: forall (E:qexp -> qexp), abstr E ->
  prog (is_qexp (Fun E))
   [All (fun x:qexp =>
          Imp (is_qexp x) (atom_ (is_qexp (E x))))] []
```

For a function expression containing `E` to be valid in Proto-Quipper, `E` should first satisfy the Hybrid `abstr` condition. This predicate guarantees that `E` encodes object-level syntax (discussed earlier and in more detail in Section 7). One of the big advantages of the Hybrid framework is that it hides such details when reasoning about OLs. The term `(Fun E)` is said to be valid if for all valid expressions `x`, the expression `E x` a is valid Proto-Quipper expression. Again, this subgoal needs to be proved intuitionistally. Proving this subgoal involves assuming that a new variable `x` is a valid expression and showing that the term obtained by replacing the bound variable in `E` by `x` is a valid expression. A similar clause can be developed for the `Let` expression, with the difference that `E` is a function of two parameters instead of one:

```
| letq: forall (E:qexp -> qexp -> qexp) (b:qexp),
  abstr (fun x => lambda (E x)) ->
  abstr (fun y => lambda (fun x => (E x y))) ->
  prog (is_qexp (Let E b))
   [All (fun x : qexp => (All (fun y:qexp =>
          Imp (is_qexp x) (Imp (is_qexp y)
                (atom_ (is_qexp (E x y)))))));
    atom_ (is_qexp b)]
   []
```

This case illustrates how `abstr` is used for functions of more than one argument. Another difference from the `Fun` case is that a second subgoal ensures the validity of the `b` argument representing the body of the `let` expression.

The last syntactic rule to present is the quantum circuit rule `circ`. Note that in the sample rules for well-formedness of Proto-Quipper expressions given above, the context Φ should contain all the free term variables; we did not include a second context for quantum variables. Following the style of the typing rule *circ*, we use the *In* and *Out* functions to identify free quantum variables and then these variables occur in the corresponding premises, but not in the conclusion. In the rule below, if Q is a set of quantum variables $q_1, \ldots q_n$ for $n \geq 0$, we write Φ_Q to abbreviate the context $is\_qexp\ q_1, \ldots, is\_qexp\ q_n$.

$$\frac{\Phi, \Phi_{Q_1} \vdash is\_qexp\ t \qquad \Phi, \Phi_{Q_2} \vdash is\_qexp\ a \qquad In(C) = Q_1 \qquad Out(C) = Q_2}{\Phi \vdash is\_qexp\ (t, C, a)}$$

This rule is encoded as the following clause of the `prog` definition.

```
| Circq: forall (C:nat) (t a:qexp), quantum_data t ->
  prog (is_qexp (Circ t C a))
   [toimpexp (FQ a) (atom_ (is_qexp a))] []
```

Recall that `quantum_data` is a predicate that holds for expressions that only involve quantum variables, the star constant, and the product constructor `Prod`. This predicate is used to directly prove the first premise $\Phi, \Phi_{Q_1} \vdash \mathit{is\_qexp}\ t$. The encoding of the second premise appears in the intuitionistic list of subgoals (the second argument to `prog`). The Coq function `FQ` returns a list of free quantum variables of an expression, and `toimpexp` is a function that takes a list quantum variables and using intuitionistic implication, adds them recursively as antecedents to a the predicate expressing well-formedness of the term. We illustrate by example:

```
toimpexp (FQ (Prod (CON Qvar 0) (CON Qvar 1)))
         (atom_ (is_qexp (Prod (CON Qvar 0) (CON Qvar 1)))) =
Imp (is_qexp (CON Qvar 0)
  (Imp (is_qexp (CON Qvar 1))
     (atom_ (is_qexp (Prod (CON Qvar 0) (CON Qvar 1))))))
```

In general, proving such an implication requires repeated applications of the $\Rightarrow$-R rule of the SL (Figure 3) in a backward direction, moving antecedents of the form *is_qexp q* into the intuitionistic context, resulting in a context that encodes Φ_{Q_2}.

As stated above, we have defined `FQ` as a Coq function. In general, users of Hybrid need to take care when defining functions with arguments whose types instantiate `expr` (defined in Section 4), which in our specific case includes the type `qexp` (defined as `expr Econ` in Section 5.1). In particular, proving properties of such functions may require exposing the de Bruijn representation of these arguments. The `FQ` function is one such example. In particular, our proofs require the following three axioms:

```
Hypothesis FQ_FUN: forall i E,
  abstr E -> FQ (Fun E) = FQ (E (Var i)).
Hypothesis FQ_LET: forall i E b,
  abstr (fun x => lambda (E x)) ->
  (forall x, proper x -> abstr (E x)) ->
  FQ (Let E b) = (FQ (E (Var i) (Var i))) ++ (FQ b).
Hypothesis FQU_LET: forall i E b,
  abstr (fun x => lambda (E x)) ->
  (forall x, proper x -> abstr (E x)) ->
  FQU (Let E b) = (FQU (E (Var i) (Var i))) ++ (FQU b).
```

These axioms are admissible, but require complex proofs. Replacing the definition of the `FQ` function with an inductive predicate that relates the input and output would solve this problem.[2]

[2] This is straightforward change, but it affects a large portion of the proof development and is left for (very near) future work.

This concludes the presentation of the rules for well-formedness of Proto-Quipper terms. Such clauses actually play two roles in proofs of OL metatheory. First, induction on well-formedness derivations is often useful, since induction directly on terms is not currently available in Hybrid. Second showing that rules are adequately represented requires these clauses. (See Section 7.)

In the following, we will present the formal typing rules that correspond to the sequent rules presented in Figure 2. Recall that the Proto-Quipper typing judgment has the form $\Phi, !\Psi; Q \vdash a : A$. The latter context Q contains all free quantum variables that appear on the right of a sequent; they appear in such a context without specifying the type since it is implicitly known to be `qubit`. The other context contains term variables and their types and has two parts where the types in $!\Psi$ contain a leading ! and the types in Φ do not. Recall also that the sequents of the SL, which we will now use to encode these rules, have the general form $\Gamma; \Delta \vdash A$, where A is a formula of the SL, Γ is an intuitionistic context of atomic formulas, and Δ is a linear context, also of atomic formulas. In our formalization, elements of the contexts of the Proto-Quipper typing judgment will be encoded as atomic formulas of the form `(typeof x A)` in the SL; in particular, the encoding of elements of $!\Psi$ will appear in the intuitionistic context Γ of the SL, while elements of the typing context Φ will appear in the linear context Δ of the SL. Each quantum variable q in Q will be placed in Δ explicitly associated with the type `qubit`, i.e., `typeof (CON (Qvar qi)) qubit`, where `qi` is the natural number encoding variable q.

We start with the ax_c rule, which we specify as two clauses depending on whether or not the first argument to the subtype relation has a leading `bang`.

```
| axc1: forall (A B:qtp) (x:qexp), validT (bang A) ->
  Subtyping A B ->
  prog (typeof x B) [atom_ (is_qexp x)] [atom_ (typeof x A)]
| axc2: forall (A B:qtp) x, Subtyping (bang A) B ->
  prog (typeof x B)
   [(And (atom_ (typeof x (bang A))) (atom_ (is_qexp x)))] []
```

In `axc1`, the `validT (bang A)` condition ensures that `A` has no leading `bang`, and as a consequence of theorem `Subtyping_Prop6`, `B` also has no leading `bang`. Thus it is required to linearly prove that `atom_ (typeof x A)`, whereas in `axc2`, it is required to intuitionistically prove `atom_ (typeof x (bang A))`. Note that for reasons of adequacy, there is the additional proof obligation to show that `x` is a valid Proto-Quipper expression (in both cases).

Regarding the ax_q and ax_x rules, as mentioned they are not encoded as clauses of `prog`. In particular, they are already covered by the initial rule of the SL (l_init of Figure 3 encoded as part of the defintion of `seq`).

The following clauses implement the two versions of the $\top$ rule, the first for $n = 0$ and the second for $n = 1$, and the *cst* rule for box^T. Similar to the well-formedness rules for constants, they are axioms and thus have no subgoals.

```
| truel: prog (typeof (CON TRUE) bool) [] []
```

```
| truei: prog (typeof (CON TRUE) (bang bool)) [] []
| box: forall T U B, valid T -> valid U ->
  Subtyping (bang (arrow (bang (arrow T U))
                         (bang (circ T U)))) B ->
  prog (typeof (CON BOX) B) [] []
```

The `prog` definition includes similar clauses for the other rules about constants and functions on circuits, i.e, the two versions of the $*_i$ and $\bot$ rules, and the *cst* rules for *unbox* and *rev*.

The next two clauses encode the λ_1 rules. Similar to the ax_c rule, we develop intuitionistic and linear versions of λ_1, depending on the type of the bound variable (whether or not it has a leading !):

```
| 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)))))]
| 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 that in both cases, the subgoal is required to be proved linearly regardless of the type of the bound variable. This is because the type of the whole expression `Fun (fun x => E x)` is linear. The difference between the two rules is the use of `lImp` when the type of the bound variable is linear, and the use of `Imp` when the type of the bound variable is duplicable. In contrast, the subgoal will be required to be proved intuitionistically for both cases of the λ_2 rule. We omit the clauses for these rules, as well as those for the $\otimes_e$ rule, since they are similar to the encoding of λ_1 presented above.

The following clause implements the *app* rule. This case requires that both expressions be available at the same time. Accordingly, it uses multiplicative conjunction (`Conj`):

```
| tap: forall E1 E2:qexp, forall T T':qtp,
  validT (arrow T T') -> prog (typeof (App E1 E2) T) []
  [(Conj (atom_ (typeof E1 (arrow T' T)))
         (atom_ (typeof E2 T')))]
```

In other words, considering backward proof from the *app* rule of Figure 2, the linear context of the conclusion must be divided into two disjoint contexts. The use of `Conj` means that the $\otimes$-R rule of Figure 3 will be used to achieve this division. Note here that there is only one *app* rule, and thus only one corresponding clause in `prog` since, whether or not `T` has a leading `bang`, the typing judgments for `E1` and `E2` must be proved linearly.

The encoding of the two $\otimes_i$ rules is similar to *app*:

```
| ttensorl: forall E1 E2:qexp, forall T T':qtp,
  validT (tensor T T') ->
  prog (typeof (Prod E1 E2) (tensor T T')) []
   [Conj (atom_ (typeof E1 T)) (atom_ (typeof E2 T'))]
| ttensori: forall E1 E2:qexp, forall T T':qtp,
  validT (bang T) -> validT (bang T') ->
  prog (typeof (Prod E1 E2) (bang (tensor T T'))) []
   [Conj (atom_ (typeof E1 (bang T)))
         (atom_ (typeof E2 (bang T')))]
```

We omit the clauses for $*_e$ and *if*, since their encoding is similar to rules already presented.

The last rule is *circ*. We encode this rule similarly to the encoding of the well-formedness rule for circuits: in this case the lists of free variables Q_1 and Q_2 have been moved in front of the turnstile symbol with the help of linear implication for `typeof` atoms, and intuitionistic implication for the `is_qexp` atoms:

```
| tCricl: forall (C:nat) (t a:qexp), forall T U,
  circIn (Crcons C) = FQ t ->
  circOut (Crcons C) = FQ a ->
  quantum_data t -> validT (circ T U) ->
   prog (typeof (Circ t C a) (circ T U))
    [And (toimp (FQ a) (atom_ (typeof a U)))
      (toimp (FQ t) (atom_ (typeof t T)))] []
| tCrici: forall (C:nat) (t a:qexp), forall T U,
  circIn (Crcons C) = FQ t ->
  circOut (Crcons C) = FQ a ->
  quantum_data t -> validT (circ T U) ->
   prog (typeof (Circ t C a) (bang (circ T U)))
    [And (toimp (FQ a) (atom_ (typeof a U)))
      (toimp (FQ t) (atom_ (typeof t T)))] []
```

Similar to the `toimpexp` presented earlier, we define `toimp` which iteratively adds well-formedness and typing assumptions for free quantum variables to a statement with the help of linear implication. Again, we illustrate by example:

```
toimp (FQ (Prod (CON (Qvar 0)) (CON (Qvar 1))))
      (atom_ (typeof (Prod (CON (Qvar 0)) (CON (Qvar 1))) A)) =
Imp (is_qexp (CON (Qvar 0)))
 (lImp (typeof (CON (Qvar 0)) qubit)
  (Imp (is_qexp (CON (Qvar 1)))
   (lImp (typeof (CON (Qvar 1)) qubit)
    (atom_ (typeof (Prod (CON (Qvar 0)) (CON (Qvar 1))) A)))))
```

It is important to note here that both the linear and duplicable versions of the circuit rule require the subgoal to be proved intuitionistically. Moreover, the subgoal is the same for both cases. Although this is in contrast with other rules, it reflects the real semantics of circuits, because if a circuit has been proven

to be of a certain type, then it should be possible to use as many instances of this circuit as needed; it is an autonomous component. If it happens that a free variable appears in a circuit construct, then this variable is of duplicable type. This semantics is imposed by the above rules since we keep the list of linear subgoals empty. According to [27], quantum variables appearing in the circuit constructs are called bounded quantum variables. That is why `FQ` of a circuit construct is supposed to return an empty list. Also, the free variables of `t` should match the input of the circuit `C`, and the free variables of `a` should match the output of `C`. The functions `circIn` and `circOut` are defined as abstract functions. In particular, [27] does not provide specific definitions for these functions. They are identified only by their domains and co-domains. In the Coq code, we define them as variables in a module. Before proving properties about a specific set of circuits, these variables must be instantiated with a particular definition. Since, like [27], we are proving only meta-level properties, we leave them abstract also.

6 Type Soundness

In this section, we formally verify the type soundness of Proto-Quipper by addressing three important properties: a type soundness under subtyping rule, inversion lemmas for Proto-Quipper *values*, and the subject reduction theorem.

6.1 Context Subtyping

In a type system with subtyping, an important soundness lemma is one that expresses that typing is preserved under the subtype relation extended to contexts. In our setting, the statement of this lemma is in a sense, a general form of the ax_c rule where the contexts in the premise and conclusion are not the same, but instead have a subtyping relation between them, in this case a kind of "contravariant" one. The lemma states, roughly, that if `seq_ il ll (atom_ A)` holds, `A` is a subtype of `B`, and the pair $(\texttt{il}', \texttt{ll}')$ is a "subtype" of the pair $(\texttt{il}, \texttt{ll})$, then `seq_ il' ll' (atom_ B)` holds. Context subtyping includes an extension of subtyping in the obvious way; an atom `typeof a t1` occurs in $\texttt{il}'$ or $\texttt{ll}'$ if and only if an atom `typeof b t2` occurs in the corresponding `il` or `ll` and `Subtyping t1 t2` holds.

Before we can make the above statement formal, we must consider additional constraints on the contexts in the sequents. As discussed in [6], in general when formalizing meta-theory, statements of theorems often relate two or more judgments and if the contexts in these judgments are non-empty, a *context relation* is often needed to specify the constraints in the form of a relation between them. The above lemma is an example that relates exactly two statements about the `seq_` predicate with their corresponding contexts. A variety of examples in a simpler setting with an intuitionistic SL are discussed in detail in [8]. We adopt this notion of context relation here, extending it to

express our requirements, which are significantly more complex. We capture both the subtyping constraints as well as the necessary additional constraints in the following inductive definition.

```
Inductive Subtypecontext :=
| subcnxt_i: Subtypecontext [] [] [] []
| subcnxt_q: forall a il il' ll ll',
  Subtypecontext il' ll' il ll ->
  Subtypecontext (is_qexp a::il') ll' (is_qexp a::il) ll
| subcnxt_iig: forall a t1 t2 il il' ll ll',
  Subtyping t1 t2 -> (exists c, t2 = bang c) ->
  Subtypecontext il' ll' il ll ->
  Subtypecontext (is_qexp a::typeof a t1::il') ll'
                 (is_qexp a::typeof a t2::il) ll
| subcnxt_llg: forall a t1 t2  il il' ll ll',
  validT (bang t1) -> validT (bang t2) ->
  Subtyping t1 t2 -> Subtypecontext il' ll' il ll ->
  Subtypecontext (is_qexp a::il') (typeof a t1::ll')
                 (is_qexp a::il) (typeof a t2::ll)
| subcnxt_lig: forall a t1 t2 il il' ll ll',
  validT (bang t2) -> (exists c, t1 = bang c) ->
  Subtyping t1 t2 -> Subtypecontext il' ll' il ll ->
  Subtypecontext (is_qexp a::typeof a t1::il') ll'
                 (is_qexp a::il) (typeof a t2::ll).
```

The additional constraints include, for example, that every time there is a typing assumption (`typeof a t`) in either a linear or intuitionistic context, there must be an assumption of the form (`is_qexp a`) in the intuitionistic context. Note that we can write `Subtypecontext il ll il ll`, where the first and third arguments are the same, and similarly for the second and fourth, when we want to ignore subtyping and care only that these additional constraints are met. We will use `Subtypecontext` for this secondary purpose rather than define a new context relation.

Before expressing and proving the central lemma mentioned above, we have to tackle a crucial theorem that is very helpful for splitting a linear context:

```
Theorem subcnxt_split: forall il il' ll ll' ll1 ll2,
  Subtypecontext il' ll' il ll -> split ll ll1 ll2 ->
  exists il1 il2 ll1' ll2',
    split ll' ll1' ll2' /\
    (forall a, In a il -> In a il1) /\
    (forall a, In a il -> In a il2) /\
    Subtypecontext il' ll1' il1 ll1 /\
    Subtypecontext il' ll2' il2 ll2.
```

This theorem explains the effect of splitting a linear context in the `Subtypecontext` relation. Recall that list splitting means dividing a list into two lists with elements in any order. This theorem helps when we are dealing with

several subgoals in `splitseq` or multiplicative conjunction goals, and we want to split the linear context without losing the benefit of the `Subtypecontext` relation. This situation is pretty common in our proofs.

We are now ready to state the central lemma.

```
Theorem subtypecontext_subtyping: forall a IL IL' LL LL' B A,
  Subtypecontext IL' LL' IL LL ->
  seq_ IL LL (atom_ (typeof a A)) -> Subtyping A B ->
  seq_ IL' LL' (atom_ (typeof a B)).
```

With the help of the above lemma and others, we successfully prove this theorem. The proof amounts to 800 lines of Coq script. The proof is by induction on the height of the sequent derivations, with cases for every possible typing rule defined in the program context `prog`.

6.2 Inversion Rules for Values

A Proto-Quipper expression is considered a *value*, or non-reducible expression, if it matches one of the following cases:

```
Inductive is_value: qexp -> Prop :=
  Varv: forall x, is_value (Var x)
| Qvarv: forall x, is_value (CON (Qvar x))
| Circv: forall a t i, quantum_data t -> quantum_data a ->
         is_value (Circ t i a)
| Truev: is_value (CON TRUE)    | Falsev: is_value (CON FALSE)
| Starv: is_value (CON STAR)    | Boxv: is_value (CON BOX)
| Unboxv: is_value (CON UNBOX)  | Revv: is_value (CON REV)
| Funvv: forall f, abstr f -> is_value (Fun f)
| Prodv: forall v w, is_value v -> is_value w ->
         is_value (Prod v w)
| Unboxappv: forall v, is_value v ->
              is_value (App (CON UNBOX) v).
```

The major role of these special expressions is in defining the language reduction rules (i.e., operational semantics), as detailed in the next section, where the main objective is to reduce a Proto-Quipper expression to one of these forms. The cases mentioned above in the definition are obvious, except for `Unboxappv`. The reason behind considering `(App (CON UNBOX) v)` a *value* is because the resulting expression is of function type; as stated in [27], the unbox operator turns a circuit into a circuit-generating function, and thus this case is similar to `Funvv`.

Now that we have the language *values*, we can prove a number of inversion lemmas (corresponding to lemmas in [27]). In each, we prove that a well typed value `v` should follow certain Proto-Quipper format(s). Here is the first example:

```
Theorem sub_one_inv: forall IL a,
  ~(In (is_qexp (CON UNBOX)) IL) -> is_value a ->
  Subtypecontext IL [] IL [] -> ~(In (is_qexp a) IL) ->
  seq_ IL [] (atom_ (typeof a one)) -> a = CON STAR.
```

In this particular inversion theorem, a value of type `one` should be the `STAR` constant. To make sure that the context is not misused, we must prevent assumptions about well-formedness of value terms from occurring in the context. To do so, we use the `Subtypecontext` relation to ensure that every `(is_qexp a)` that appears in `IL` is associated with a typing atom `(typeof a A)` in `IL` This way, if `~(In (is_qexp a) IL)` is assumed in the inversion theorem then no typing hypothesis for the value "`a`" appears in `IL`. We add a similar condition for the `UNBOX` due to the fact that `(App (CON UNBOX) v)` is a value: if `UNBOX` is assumed in `IL` to be of type `(arrow A one)` (which is a false assumption according to Proto-Quipper typing rules), then by assuming the existence of a value `v` of type `A`, one can prove that "`a`" has the form `(App (CON UNBOX) v)`, which is wrong. The conclusion of the above theorem is also valid for the type `(bang one)`, which is also formally proved. Similar results have been proved for all other values. To avoid repetition, we pick one more interesting example to illustrate, however, the reader still can find a full list of the inversion rules online at [15].

The following theorem is concerned with the values of the `arrow` type constructor. In contrast to the previous theorem (and all other cases), we have several possibilities that satisfy this typing format:

```
Theorem sub_bangarrow_inv: forall IL LL a T U,
  ~(In (is_qexp (CON UNBOX)) IL) -> valid T -> valid U ->
  (forall v, a = App (CON UNBOX) v -> ~(In (is_qexp v) IL)) ->
  is_value a -> Subtypecontext IL LL IL LL ->
  ~(In (is_qexp a) IL) ->
  seq_ IL LL (atom_ (typeof a (bang (arrow T U)))) ->
  (exists f, a = Fun f) \/ (exists T0, (a = CON (BOX T0)) \/
  (a = CON UNBOX) \/ (a = CON REV) \/
  (exists t i u, a = App (CON UNBOX) (Circ t i u)).
```

According to above theorem, the value "`a`" would be a function, a quantum circuit conversion function, or a function application over the `UNBOX` constant.

The proofs of the above inversion rules are done by case analysis of the possible values. For each case, it is proved that `seq_ IL LL (atom_ (typeof a A))` requires that `(In (typeof a A) IL)` or `(In (typeof a A) LL)`, which leads to a contradiction except for the particular value we are interested in. For instance, in the theorem `sub_one_inv`, assuming that `seq_ IL LL (atom_ (typeof a one))` leads to a contradiction in all cases except for the `STAR` constant.

We also prove inversion rules that serve the opposite purpose from the above ones, where the expression format is provided and the theorems conclude the right corresponding type and/or typing rules. Those rules are more general, i.e., they are not restricted to values. For instance, the following theorem characterizes the typing rules for the `Slet` statement:

```
Theorem sub_slet_inv: forall IL LL a b A,
  Subtypecontext IL LL IL LL ->
  seq_ IL LL (atom_ (typeof (Slet a b) A)) ->
  ~(In (is_qexp (Slet a b)) IL) ->
  (exists B, Subtyping B A /\
   (splitseq_ IL LL
     [Conj (atom_ (typeof a B)) (atom_ (typeof b (bang one)))] \/
    splitseq_ IL LL
     [Conj (atom_ (typeof a B)) (atom_ (typeof b  one))])) \/
  (validT A /\
   (splitseq_ IL LL
     [Conj (atom_ (typeof a A)) (atom_ (typeof b (bang one)))] \/
    splitseq_  IL LL
     [Conj (atom_ (typeof a A)) (atom_ (typeof b one))])).
```

The `Subtypecontext` is used here again to ensure that we are dealing with a valid typing context as explained earlier; it has nothing to do with the subtyping between contexts since the super-context and subcontext are the same. The above theorem concludes two possible cases for a well-typed `Slet` expression. First, the right side of the main disjunction corresponds to the case when the type `A` is deduced directly from the `SLet` typing rules (see the `tsletl` and `tsleti` of the `Prog` definition in [15]). Second, the left disjunct corresponds to the case when the typing deduction has gone through one or more subtyping rules (see rules `axc1` and `axc2` of the `prog` definition discussed above and in [15]) followed by the regular `SLet` typing rules).

Here is another example for the circuit construct:

```
Theorem sub_Circ_inv: forall IL LL t a c A,
  Subtypecontext IL LL IL LL ->
  seq_ IL LL (atom_ (typeof (Circ t c a) A)) ->
  ~(In (is_qexp (Circ t c a)) IL) ->
  (exists T T' B,
    Subtyping B A /\ validT (circ T T') /\ LL = [] /\
    splitseq_ IL []
     [And (toimp (FQ a) (atom_(typeof a T')))
          (toimp (FQ t) (atom_(typeof t T)))] /\
    (B = circ T T' \/ B = bang (circ T T'))) \/
  (exists T T',
    validT (circ T T') /\ LL = [] /\
    splitseq_ IL []
     [And (toimp (FQ a) (atom_(typeof a T')))
          (toimp (FQ t) (atom_(typeof t T)))] /\
    (A = circ T T' \/ A = bang (circ T T'))).
```

An interesting fact about a well-typed circuit construct is that its linear context should be empty, and this is expected as the circuit is supposed to be used several times, and hence it does not depend on linear typing atoms. Re-

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

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

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

Fig. 4 If statement reduction rules

call that the quantum variables that appear in "`t`" and "`a`" are considered bound variables.

Similar results have been proved for other Proto-Quipper expressions, e.g., functions, function application, circuit conversion constants, etc.

6.3 Subject Reduction

Reduction rules, i.e., operational semantics, are crucial for any programming language, defining how valid expressions in a language are simplified until a non-reducible expression is reached, i.e. a value. In the context of a language's reduction rules, it is very important to ensure the integrity of typing rules by ensuring that a reduction of a well-typed expression should not affect the expression's type (i.e., type soundess). In Proto-Quipper, there are seventeen reduction rules, which handle all possible cases while ensuring that only one reduction rule can be applied at a time. To keep the description concise, we choose five reduction rules to explain in this paper, which are good representative samples. Similar to typing rules, we encode Proto-Quipper reduction rules as part of the inductive definition of `prog`. For the complete set of rules, we refer the reader to [27], and for their complete encoding in Coq, see [15].

Given a closure pair $[C, a]$, where C is a circuit constant and a is a Proto-Quipper expression such that $FQ(a) \in Out(C)$, an `if` statement is reduced according to the rules listed in Figure 4. The formal Coq presentation of the rules in Figure 4 as part of the `prog` definition is 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 `valid_c` ensures that `[C',(If b' a1 a2)]` forms a valid closure. Note that the rule `ifr` is only applicable if "`b`" is not a value, otherwise this rule

$$\frac{[D,a] \to [D',a']}{[C,(t,D,a)] \to [C,(t,D',a')]}\ circ$$

$$\frac{Spec_{FQ(v)}(T) = t \qquad New(FQ(t)) = D}{[C, box(v)] \to [C,(t,D,v\ t)]}\ box$$

$$\frac{bind(u,v) = b \qquad Append(C,D,b) = (C',b')}{[C,(unbox\ (u,D,u'))\ v] \to [C',b'(u')]}\ unbox$$

Fig. 5 Circuit reduction rules

would be applied an infinite number of times without achieving any progress. When "b" is a value, i.e., $True$ or $False$ one of the other two rules can be applied.

The remaining examples of Proto-Quipper reduction rules that we consider here are the circuit construct and its conversion function boxing and unboxing rules, which appear in Figure 5. $Spec_Q$ is a function that given a quantum data type T, returns a *specimen* t of that type, i.e., a quantum data term having type T that has the further property that all quantum variables in t are "fresh" with respect to the quantum variables occuring in Q. New is a function that creates a new identity circuit, i.e., given a set of quantum variables, these variables serve as the inputs and outputs, and the new circuit contains only wires where inputs are mapped directly to the same outputs.

The notion of a *binding* expresses the way in which wires should be connected. The *bind* function is a bijection on quantum variables, where it wires each quantum variable in u with the corresponding one in v. For example, $bind(\langle q_1, q_2\rangle, \langle q_3, q_4\rangle) = \{(q_1, q_3); (q_2, q_4)\}$. The *Append* function is responsible for appending two circuits. It returns the new circuit and a binder to rename the outputs of the newly created circuit. This rename step is not really required since the newly created circuit can use the same names of the output of the circuit D as its output; however, it is considered an added feature in the Proto-Quipper language. The following are the corresponding formal reduction rules of the *box* and *unbox* rules listed in Figure 5:

```
| boxr: forall v T t C, valid T -> is_value v ->
  t = (Spec (newqvar v) T) ->
  prog (reduct C (App (CON (BOX T)) v) C
         (Circ t (circNew (FQ t)) (App v t)))
    [atom_ (is_qexp  v)] []
| unboxr: forall u u' v C C' D b',
  quantum_data u -> quantum_data u' -> quantum_data v ->
  bind u v -> bind u' b' ->
  circApp (Crcons C) (Crcons D) = C' ->
  prog (reduct (Crcons C)
               (App (App (CON UNBOX) (Circ u D u')) v) C' b')
    [atom_ (is_qexp v); atom_ (is_qexp b')] []
```

In the `boxr` clause, the function `newqvar` returns a variable that is fresh with respect to its argument, which is a quantum term. In our implementation of

the `Spec` function, the first argument is a natural number and all quantum variables in the result are represented by numbers greater than this input number. Together `Spec` and `newqvar` implement the *Spec* function in the *box* rule.

In our formalization, we implement a relation (`bind u v`), which holds if `u` and `v` are related by some binding, i.e., they are the same quantum data terms up to the renaming of quantum variables. This relation is used twice in the `unboxr` clause to implement the binding relations in the *unbox* rule. We note that the *Append* function is formalized as an abstract function without concrete implementation, where Proto-Quipper does not provide one; rather it provides its properties which we axiomatize in the above rule.

Similar rules have been formalized for the rest of Proto-Quipper expressions and can be found in [15].

Now, we can present the most important property of the Proto-Quipper language, namely subject reduction, which completes the type soundness result of the Proto-Quipper type system and its operational semantics:

```
Theorem subject_reduction: forall i IL C C' a a' LL1 LL2 A,
  (forall V, In V (get_boxed a) ->
    ~(exists n, V = (Var n) \/ V = (CON (Qvar n)))) ->
  NoDup (FQUC a') -> NoDup (FQU a') ->
  (forall t, In t IL ->
    (exists  n, t = is_qexp (Var n) \/
                t = is_qexp (CON (Qvar n)) \/
                exists T, t = typeof (Var n) T)) ->
  (forall q, In q (FQ a') -> In (typeof q qubit) LL2) ->
  Subtypecontext IL LL1 IL LL1 ->
  Subtypecontext IL LL2 IL LL2 ->
  common_ll a a' LL1 LL2 ->
  seq_ i IL [] (atom_ (reduct C a C' a')) ->
  exists j, seq_ j IL LL1 (atom_ (typeof a A)) ->
  exists k, seq_ k IL LL (atom_ (typeof a' A)).
```

The three underlined predicates represent the core of the the subject reduction theorem. The three predicates translate the typical meaning of the subject reduction: *an expression `a'`, that is the reduction of a well-typed expression `a`, is also well-typed and hold the same type as the original expression `a`.*

The remaining hypotheses belong to two categories and require explanation. The three lines following the theorem name are additional well-formedness constraints. As we will describe below, the one involving `get_boxed` was unexpected and illustrates the merits of formal proofs to check paper-and-pencil ones, which often overlook the kind of detail captured by this definition. The remaining lines are conditions on contexts. We are able to reuse our `Subtypecontext` definition to capture some of the requirements, but in addition, we define an additional context relation `common_ll` as well as add some additional requirements on the form of assumptions appearing in contexts.

In general, type soundness results are often stated and proved for typing of closed terms and empty contexts. Here, we could restrict terms so that they contain no free term variables (in particular, require that `a` and `a'` contain no free term variables), but we cannot do the same for quantum variables since there is no lambda binder for quantum variables in the language. Most of our additional hypotheses provide the required structure for the appearance of assumptions about quantum variables in contexts, but we go beyond that and also allow free term variables in `a` and `a'`. Doing so requires some additional hypotheses in the statement of the theorem, but allows us to prove a more general form. The version requiring `a` and `a'` to be closed follows as a corollary. In summary, the basic requirements for free quantum variables are that for every variable `q` (of the form `(CON (Qvar n))`), `q` is free in `a` or `a'` if and only if there is an assumption `(is_qexp q)` in `IL`, `q` is free in `a` if and only if there is an assumption of the form `(typeof q qubit)` in `LL1`, and `q` is free in `a'` if and only if there is an assumption of the form `(typeof q qubit)` in `LL2`.

We explain the hypotheses in more detail one by one in the following.

- ```
  (forall t, In t IL ->
    (exists  n, t = is_qexp (Var n) \/
                t = is_qexp (CON (Qvar n)) \/
                exists T, t = typeof (Var n) T)):
  ```
  This hypothesis restricts the intuitionistic context so that it includes only typing and well-formedness assumptions about term variables and quantum variables. Note that the inversion rules `sub_one_inv`, `sub_bangarrow_inv`, `sub_slet_inv`, and `sub_Circ_inv` in Section 6.2 all restrict the form of the argument to `is_qexp`, which is the expression subject to inversion, when it appears in the intuitionistic context. In these theorems, the restrictions are carried over to the linear contexts because of the way the contexts are related according to the `Subtypecontext` predicate. The same applies here.
- `Subtypecontext IL LL1 IL LL1` and `Subtypecontext IL LL2 IL LL2`: These predicates ensures that the constraints discussed in Section 6.1 about the relationship between the intuitionistic and linear contexts are met. For example, each typing assumption of the form `(typeof a t)` in any context must be associated with a well-formedness assumption `(is_qexp a)` in the intuitionistic context. In addition, it restricts the linear context to contain `typeof` assumptions only.
- `common_ll a a' LL1 LL2`:
  One might wonder why the theorem requires two different linear contexts (`LL1` and `LL2`); in particular, the common subject reduction for classical OLs maintains the same context in all judgments. The answer is *quantum variables*. Proto-Quipper reduction rules allow for quantum variable renaming (i.e., in the case of circuit appending), and allow introduction of new quantum variables (e.g., a circuit with zero inputs that produces quantum variables at the output, which typically represents an initialization circuit). As a result, the linear context of the original expression is
  ```

usually a bit different from the context for the reduced one; they are similar modulo quantum variables. That is why we have defined the predicate `common_ll`, which ensures that all typing assumptions in `LL1` and `LL2` are the same except for quantum variables, where quantum variables in `LL1` belong to the set of free quantum variables of `a`, and quantum variables in `LL2` belong to the set of free quantum variables of `a'`.

– `(forall q, In q (FQ a') -> In (typeof q qubit) LL2)`:
The previous predicate does not assure that `LL2` contains assumptions about all free quantum variables of `a'`; it only ensures if a quantum variables exists in `LL2` then it belongs to `(FQ a')`. That is why we added this assumption. One might ask why not to do the same for `LL1`; the answer that we do not need that assumption since `a` is well-typed under `LL1` and hence it must contain all quantum variable of `(FQ a)`. This fact is proved as per the following theorem (note the equivalence in the last two lines):

```
Theorem LL_FQ: forall u A IL LL,
  (forall q T,
    In (typeof (CON (Qvar q)) T) LL -> T = qubit) ->
  (forall q T, In (typeof (CON (Qvar q)) T) LL ->
    count_occ eq_dec LL (typeof (CON (Qvar q)) T) = 1) ->
  (forall t, In t IL ->
    (exists  n, t = is_qexp (Var n) \/
                t = is_qexp (CON (Qvar n)) \/
                exists T, t = typeof (Var n) T)) ->
  Subtypecontext IL LL IL LL ->
  seq_ IL LL (atom_ (typeof u A)) ->
  (forall q, In (CON (Qvar q)) (FQ u) <->
             In (typeof (CON (Qvar q)) qubit) LL).
```

– `NoDup (FQUC a')` and `NoDup (FQU a')`:
`FQU` is similar to `FQ` in that it returns the free quantum variables occurring in a term. The difference is that `FQ` returns a set, while `FQU` returns a list so that if there is more than one occurrence of a free variable in a term, it occurs more than once in the output list. `FQUC` is similar, but also includes bounded quantum variables of quantum circuits. (In `FQ` and `FQU`, those variables are excluded.) The predicate `NoDup` comes from Coq's list library and is true of lists that contain no duplicates. In quantum calculus, quantum variables are processed once. We use `FQU` to rule out expressions that involve quantum replicas. `FQUC` ensures no quantum variable replicas inside circuit bodies.

– `(forall V, In V (get_boxed a) ->`
`~(exists n, V = (Var n) \/ V = (CON (Qvar n))))`:
The function `get_boxed` returns a list of all expression that are arguments to the box^T operator, i.e., for the expression `(APP (CON (BOX T)) a)`, it returns `a`. We want to make sure that those expressions are neither free variables nor quantum variables. Unlike the other hypotheses in the theorem, which are mostly due to our method of formalization, this condition

should be respected in the language implementation itself; otherwise the language allows for expressions that are not *values* and there is no reduction rule that applies. Without this, neither subject reduction nor progress properties hold for Proto-Quipper.

The proof of the above theorem involves several lemmas and theorems and makes significant use of a number of the inversion rules discussed in the previous section. It is proved by induction on `i`, the height of the proof of the sequent containing the typing judgment, followed by a case analysis of 17 reduction rules, each of which require at least 10 major subgoals due to case analysis and inversions.

This concludes our formalization of Proto-Quipper in Hybrid, along with our development of a generic linear specification logic used as a platform to reason about quantum programming languages. In the following section, we will discuss a very critical aspect of our development where we ensure that the developed formalization models the intended behavior of Proto-Quipper as presented in [27].

7 Adequacy

It is important to show that both syntax and inference rules are adequately represented in Hybrid. Adequacy of syntax encoding, also called *representational adequacy*, is discussed for the lambda calculus as an OL in Hybrid in [2] and proved in detail in [4], while adequacy for a fragment of a functional programming language known as Mini-ML is proved in [7]. Proto-Quipper contains the lambda calculus as a sublanguage, and representational adequacy for the full language is a straightforward extension of these other results. We give a brief overview here.

First, the `proper` and `abstr` predicates, defined in the Hybrid library, provide important tools for proving representational adequacy of any OL. As mentioned, the `proper` predicate rules out terms with dangling indices, which clearly cannot occur in well-formed terms of any OL. The `abstr` predicate, as mentioned, applies to arguments of the `lambda` operator. Recall that this operator has one argument of functional type (`expr` $\rightarrow$ `expr`). To adequately represent OL syntax, we must rule out *exotic* functions, i.e., functions that do not encode OL lambda terms. To illustrate, we consider the example from [7]. Suppose we have (`lambda` e), where $e = (\lambda x.$`count` $x)$ where (`count` x) counts the total number of variables and constants occurring in x. This function clearly does not represent syntax. Only functions that behave *uniformly* or *parametrically* on their arguments, such as the one discussed in the introduction, represent OL terms. The `abstr` predicate identifies exactly this set of functions. See [5] and the related work section of [7] for a careful analysis of this phenomenon. The former applies specifically in the context of Coq. Also, see [15] for the precise definitions of these two predicates.

Second, proving representational adequacy requires defining an encoding function between OL terms and their representation in Hybrid, and showing

that this function is a bijection. In particular, we write $\varepsilon_\Gamma(\cdot)$ for the encoding function from Proto-Quipper terms with free term variables in Γ to terms of type `qexp`. We omit its definition, but note that it maps each term variable x in Γ to a distinct Coq variable $\mathtt{x} : \mathtt{qexp}$, and each quantum variable q in Γ to an expression of the form $(\mathtt{CON}\ (\mathtt{Qvar}\ q_i))$, where q_i is a distinct natural number for each quantum variable. Consider the judgment $\Gamma \vdash \mathit{is\_qexp}\ a$ and a full set of inference rules for this judgment in the style of those presented in Section 5.2. Let $\{x_1, \ldots, x_n\}$ be the set of term variables in Γ and let $\mathtt{x_1}, \ldots, \mathtt{x_n}$ be the encodings of these variables. We must prove that for any Proto-Quipper term a, if $\Gamma \vdash \mathit{is\_qexp}\ a$, then the following is provable in Coq:

```
proper x₁ → ⋯ proper xₙ →
exists (i : nat),
  seq i [(is_qexp x₁); …; (is_qexp xₙ)] []
      (atom_ (is_qexp ε_Γ(a)))
```

Furthermore, we write $\delta_\Gamma(\cdot)$ for the decoding function. Let `Γ` be Coq variables and terms $\{\mathtt{x_1}, \ldots, \mathtt{x_n}, (\mathtt{CON}\ (\mathtt{Qvar}\ q_{i_1})), \ldots, (\mathtt{CON}\ (\mathtt{Qvar}\ q_{i_m}))\}$ of type `qexp`, let `E` be a term of type `qexp`, and let Γ be:

$$\{\delta_\Gamma(\mathtt{x_1}), \ldots, \delta_\Gamma(\mathtt{x_n}), \delta_\Gamma(\mathtt{CON}\ (\mathtt{Qvar}\ q_{i_1})), \ldots, \delta_\Gamma(\mathtt{CON}\ (\mathtt{Qvar}\ q_{i_m}))\}.$$

We must prove that if the following is provable in Coq:

```
proper x₁ → ⋯ proper xₙ →
exists (i : nat),
  seq i [(is_qexp x₁); …; (is_qexp xₙ)] [] (atom_ (is_qexp E))
```

then $\delta_\Gamma(\mathtt{E})$ is defined and yields a Proto-Quipper term a such that $\Gamma \vdash \mathit{is\_qexp}\ a$. Additionally, $\varepsilon_\Gamma(\delta_\Gamma(\mathtt{E})) = \mathtt{E}$ and $\delta_\Gamma(\varepsilon_\Gamma(a)) = a$. The above results correspond to Lemma 21 (called *Validity of Representation*) and Lemma 22 (*Completeness of Representation*), respectively, in [7].[3]

Proving the adequacy of the encoding of inference rules is similar. Note that in the above statements, well-formedness was expressed as a set of inference rules of a judgment of the form $\Gamma \vdash \mathit{is\_qexp}\ a$, and the proofs of these statements require showing a bijection between proofs (on paper) using the inference rules of this judgment, and proofs using our encodings of `prog` and `seq`. The same approach is used to prove the adequacy of the other two judgments $\Gamma; Q \vdash a : A$ and $[C, a] \to [C', a']$. Additionally, for all judgments except well-formedness, *internal adequacy* lemmas must be proven. Such lemmas correspond to Lemma 20, clauses 2 and 4 for the reduction and typing judgments, respectively, of MiniML in [7].[4] They are called internal adequacy lemmas because they can be formalized in Hybrid and they are an important part of the general adequacy proofs for these judgments. In general, such properties state that whenever an OL judgment that is defined as part of the definition

[3] We also impose the restriction that the Coq derivation must be *minimal* in the same sense as described there. See [7] for details.

[4] A variety of other internal adequacy lemmas are shown in [8].

of `prog` can be proved, then the OL terms in this judgment are well-formed. The `abstr` conditions are important for proving these lemmas. The following theorem expresses internal adequacy for the Proto-Quipper typing judgment.

```
Lemma hastype_isterm_ctx :
  forall (M:qexp) (T:qtp) (iq it lt:list atm),
  ctxR iq it lt ->
  seq_ it lt (atom_ (typeof M T)) ->
  seq_ iq [] (atom_ (is_qexp M)).
```

Note that it uses a context relation `ctxR` relating the intuitionistic and linear contexts of the `typeof` predicate to the intuitionistic context of the `is_qexp` predicate. (The linear context is always empty when proving well-formedness of terms.) The definition of this relation is as follows:

```
Inductive ctxR: list atm -> list atm -> list atm -> Prop :=
| nil_cr: ctxR nil nil nil
| cons_q_cr: forall (iq it lt:list atm) (x:qexp),
  proper x -> ctxR iq it lt ->
  ctxR (is_qexp x::iq) (is_qexp x::it) (lt)
| cons_l_cr: forall (iq it lt:list atm) (x:qexp)
  (T:qtp),  proper x -> ctxR iq it lt ->
  ctxR (is_qexp x::iq) (is_qexp x::it) (typeof x T::lt)
| cons_i_cr: forall (iq it lt:list atm) (x:qexp)
  (T:qtp), proper x -> ctxR iq it lt ->
  ctxR (is_qexp x::iq) (typeof x T::is_qexp x::it) lt.
```

In addition the following two lemmas are needed. The first is a standard lemma about intuitionistic contexts as found in the examples in [8]. The second is unique to linear contexts, and thus is new.

```
Lemma qexp_strengthen_weaken:
  forall (M:qexp) (Phi1 Phi2:list atm),
  (forall (M:qexp),
    In (is_qexp M) Phi1 -> In (is_qexp M) Phi2) ->
  seq_ Phi1 [] (atom_ (is_qexp M)) ->
  seq_ Phi2 [] (atom_ (is_qexp M)).

Theorem ctxRconcat: forall iq it lt lt1 lt2,
  ctxR iq it lt ->  lt = lt1++lt2 ->
  exists it1 it2,
    (forall a, In a it -> In a it1) /\
    (forall a, In a it -> In a it2)  /\
    ctxR iq it1 lt1 /\ ctxR iq it2 lt2.
```

Note that the conclusion of the `hastype_isterm_ctx` lemma only involves the term `M`. No similar conclusion is required for `T` because Proto-Quipper types are defined directly as an inductive type in Coq. There are no binders

in types, and thus the correspondence between terms of type `T` and types of Proto-Quipper is direct.

We have proved a similar result for the reduction rules, showing that the language's operational semantics does not lead to invalid expressions (see [15]).

8 Conclusion

We have presented our formalization of Proto-Quipper in Hybrid. This work involved encoding a linear specification logic and carrying out a large case study in Hybrid, in the sense that we encode and reason about the complete Proto-Quipper specification, the most complex OL considered so far, and we prove type soundness, one of the central results in [27].

In order for Hybrid with a linear SL to become a fully operational logical framework, we will need to provide suitable automation of proofs, based on lessons learned from this case study. Adding such automation is an important direction for future work.

Another important direction is extending the formalization to other more complex properties and to other quantum programming languages. For example, the other main result [27] is a progress theorem for Proto-Quipper, which is an obvious next step for us. We do not foresee any difficulty, though its proof will likely be as long and detailed as the proof of type soundness. Also, it should be straightforward to adapt the existing formal proofs to new versions of Proto-Quipper. For example, a new version called Proto-Quipper-M is introduced in [26]. This version does not have subtyping, which should mean that certain aspects of formal proofs of its metatheory will be easier. In the future, we hope to use our system as an environment in which new Proto-Quipper metatheory can be simultaneously developed and formalized as versions of the language evolve.

It should also be possible to directly extend our work from Proto-Quipper, which is fairly expressive though not Turing complete, to the full Turing-complete Quipper language. Similarly, we expect to be able to extend this work directly in order to apply it to other Turing-complete quantum lambda calculi such as those introduced by Zorski et. al. [13,14,31] and Grattage et. al. [1, 9]. As another example, the QWIRE language in [25] 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. In our framework, it would be interesting to also study the meta-theory of this language.

A variety of other logical frameworks implementing linear logic have been developed. The ordered linear logic (OLF) mentioned earlier [24] is one such example, where type soundness for a *continuation*-based abstract machine for the functional programming language Mini-ML (an OL that is simpler than Proto-Quipper) is proven [7] following the statement in [3]. Another example of OLs that benefit from a framework based on linear logic are those with

imperative features. A common thread of such examples is that they contain the notion of updatable state, which can be handled fairly directly by the linear features of the framework. The case study we present in this paper is the first one in Hybrid where the OL itself is linear, in the sense that the linear lambda calculus forms the core of Proto-Quipper.

Examples involving mutable state have motivated a variety of proposals for frameworks based on linear logics that support HOAS. For other examples that benefit from linear features, see the overview in [20]. Examples of frameworks proposed include Lolli [12], Forum [19], and LLF [3]. The logical framework LF [11] and its implementation in Twelf represents one of the earliest logical frameworks supporting HOAS and based on minimal intuitionistic logic. LLF is a conservative extension of LF with multiplicative implication, additive conjunction, and unit.

Type soundness proofs for various OLs has been a common benchmark for logical frameworks supporting HOAS and implementing an intuitionistic logic, starting with some of the earliest logical frameworks like LF [30]. In [17], Mini-ML with mutable references, an imperative version of the Mini-ML language mentioned above, is studied in Hybrid. Five versions of type soundness are proved using three different SLs, intuitionistic, linear, and OLF as implemented in [7], and their formalizations compared.

Acknowledgements

The authors would like to thank Julien Ross and Peter Selinger for useful discussions on technical details as well as on approaches and directions for this work. We would also like to thank the reviewers for useful comments for improving this paper.

References

1. Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 249–258. IEEE, 2005.
2. Simon Ambler, Roy L. Crole, and Alberto 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, pages 13–30. Springer, 2002.
3. Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Compututation*, 179(1):19–75, 2002.
4. Roy L. Crole. The representational adequacy of Hybrid. *Mathematical Structures in Computer Science*, 21(3):585–646, 2011.
5. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 124–138. Springer, 1995.
6. Amy Felty, Alberto Momigliano, and Brigitte Pientka. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Mathematical Structures in Computer Science*, pages 1–34, 2017.

7. Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.
8. Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2—a survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015.
9. Jonathan Grattage. An overview of QML with a concrete implementation in haskell. *Electronic Notes in Theoretical Computer Science*, 270(1):165–174, 2011.
10. Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Thirty-Fourthh ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–342. ACM, 2013.
11. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
12. Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Compututation*, 110(2):327–365, 1994.
13. Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. On a measurement-free quantum lambda calculus with classical control. *Mathematical Structures in Computer Science*, 19(2):297–335, 2009.
14. Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.
15. Mohamed Yousri Mahmoud and Amy P. Felty. Formalization of metatheory of the Quipper quantum programming language in a linear logic: Coq script. `http://www.site.uottawa.ca/~afelty/jar-submission`.
16. Mohamed Yousri Mahmoud and Amy P. Felty. Formal meta-level analysis framework for quantum programming languages. In *12th Workshop on Logical and Semantic Frameworks with Applications (LSFA 2017)*, Electronic Notes in Theoretical Computer Science, 2018. to appear.
17. Alan J. Martin. *Reasoning Using Higher-Order Abstract Syntax in a Higher-Order Logic Proof Environment: Improvements to Hybrid and a Case Study*. PhD thesis, University of Ottawa, 2010.
18. Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.
19. Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.
20. Dale Miller. Overview of linear logic programming. In Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Notes*, pages 119–150. Cambridge University Press, 2004.
21. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
22. Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31(3es):1–6, 1999. Article No. 11.
23. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.
24. Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.
25. Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. In *Postproceedings of the 14th International Conference on Quantum Physics and Logic (QPL 2017)*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 119–132, 2018.
26. Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. In *Postproceedings of the 14th International Conference on Quantum Physics and Logic (QPL 2017)*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 164–178, 2018.
27. Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, August 2015. arXiv:1510.02198 [quant-ph].

28. Peter Selinger. Personal communication, January 2016.
29. Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classial control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.
30. The Twelf Project. Introduction to Twelf: Proving metatheorems about the STLC. `http://twelf.org/wiki/Proving\_metatheorems:Proving\_metatheorems\_about\_the\_STLC`, 2009. Accessed: 2016-10-1.
31. Margherita Zorzi. On quantum lambda calculi: a foundational perspective. *Mathematical Structures in Computer Science*, 26(7):1107–1195, 2016.