On Dynamic Lifting and Effect Typing in Circuit Description Languages

Andrea Colledan
University of Bologna, Italy
INRIA Sophia Antipolis, France

Ugo Dal Lago
University of Bologna, Italy
INRIA Sophia Antipolis, France

Abstract

In the realm of quantum computing, circuit description languages represent a valid alternative to traditional QRAM-style languages. They indeed allow for finer control over the output circuit, without sacrificing flexibility nor modularity. We introduce a generalization of the paradigmatic lambda-calculus Proto-Quipper-M, which models the core features of the quantum circuit description language `Quipper`. The extension, called Proto-Quipper-K, is meant to capture a very general form of dynamic lifting. This is made possible by the introduction of a rich type and effect system in which not only *computations*, but also the very *types* are effectful. The main results we give for the introduced language are the classic type soundness results, namely subject reduction and progress.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Type theory; Hardware → Quantum computation

Keywords and phrases Circuit-Description Languages, λ-calculus, Dynamic lifting, Type and effect systems

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.3

Related Version *Extended Version*: `https://arxiv.org/abs/2202.07636` [2]

Funding The research leading to these results has received funding from the European Union - NextGenerationEU through the Italian Ministry of University and Research under PNRR - M4C2 - I1.4 Project CN00000013 "National Centre for HPC, Big Data and Quantum Computing".

1 Introduction

Despite the undeniable fact that large-scale, error-free quantum hardware has yet to be built [19], research into programming languages specifically designed to be compiled towards architectures including quantum hardware has taken hold in recent years [17]. Most of the proposals in this sense (see [9, 22, 24] for some surveys) concern languages that either express or can be compiled into some form of *quantum circuit* [15], which can then be executed by quantum hardware. This reflects the need to have tighter control over the kind and amount of quantum resources that programs employ. In this scenario, the idea of considering high-level languages that are specifically designed to *describe* circuits and in which the latter are treated as first-class citizens is particularly appealing.

A typical example of this class of languages is `Quipper` [10, 11], whose underlying design principle is precisely that of enriching a very expressive and powerful functional language like `Haskell` with the possibility of manipulating quantum circuits. In other words, programs do not just build circuits, but also treat them like data, as can be seen in the example from Figure 1. `Quipper`'s meta-theory has been studied in recent years through the introduction of a family of research languages that correspond to suitable `Quipper` fragments and extensions,

```
alice ::  Qubit -> Qubit -> Circ (Bit,Bit)
alice q a = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 return (x,y)
```

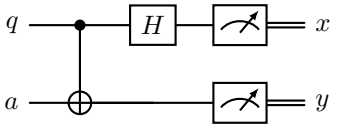


Figure 1 Alice's part of the quantum teleportation circuit. The `Quipper` program on the left builds the circuit on the right, but while doing so it also manipulates the smaller circuit `qnot a`, enriching it with control.

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b q a = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 (u,s) <- dynamic_lift (x,y)
 b <- if s then gate_X b else return b
 b <- if u then gate_Z b else return b
 return b
```

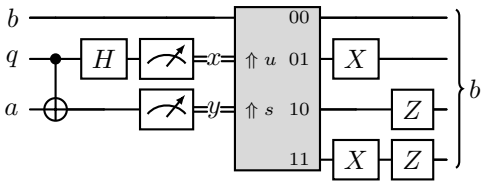


Figure 2 Quantum teleportation circuit with dynamic lifting. The gray box is not a gate, but rather represents the dynamic lifting of bit wires x and y into variables u and s and the extension of the circuit with one of four possible continuations for the remaining wire b, depending on the outcome of the intermediate measurements.

which usually take the form of linear λ-calculi. We are talking about a family of languages whose members include Proto-Quipper-S [21], Proto-Quipper-M [20], Proto-Quipper-D [5, 7], Proto-Quipper-L [13] and Proto-Quipper-Dyn [6].

An aspect that until very recently has only marginally been considered by the research community is the study of the meta-theory of so-called *dynamic lifting*, i.e. the possibility of allowing the (classical) value flowing in one of the wires of the underlying circuit, naturally unknown at circuit building time, to be *visible* in the host program for control flow. As an example, one could append a unitary to some of the wires *only if* a previously performed measurement has yielded a certain outcome. This is commonly achieved in many quantum algorithms via classical control, but Quipper also offers a higher-level solution precisely in the form of dynamic lifting, as can be seen in the example program in Figure 2. Notably, such a program cannot be captured by any of the calculi in the Proto-Quipper family, with the exception of Proto-Quipper-L [13] and Proto-Quipper-Dyn [6], arguably the most recent additions to the family.

Looking at the `Quipper` program in Figure 2, one immediately realizes that the two branches of both occurrences of the `if` operator change the underlying circuit in a uniform way, i.e. the number and type of the wires are the same in either branch. What if, for instance, we wanted to condition the execution of a measurement on a lifted value, like in Figure 3? Unfortunately, `Quipper` does *not* allow the program in Figure 3 to be typed, on account of the two branches of the `if` operators having different types. A program such as this could come from the so-called *measurement calculus* [4], where it would be referred to as a *pattern*, i.e. a sequence of simple operations that act on qubits in a way not so different from what happens in quantum circuits, but with the crucial difference that the basis of a

```
oneWay ::  Qubit -> Qubit -> Bit
oneWay q a = do
 q <- hadamard q
 x <- measure q
 u <- dynamic_lift x
 a <- if u then meas_alpha a else return a
 a <- if not u then meas_beta a else return a
 return a
```

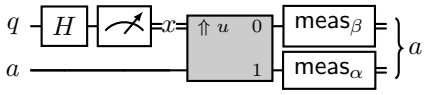


Figure 3 An example of conditional measurement, where meas_α and meas_β measure a qubit in two distinct bases. This program is ill-typed in `Quipper`.

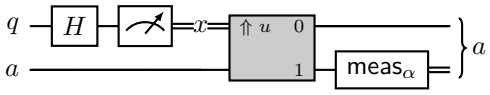


Figure 4 The circuit produced by the program in Figure 3 without the last conditional.

measurement can depend on the outcome of previous measurements. Globally, the program builds a uniformly typed circuit (which always outputs a bit), but locally there are some intermediate steps where the output type is heterogeneous. Of course, in this case, the two `if` operators could be collapsed into one, avoiding local heterogeneity, but there are cases in which keeping two conditionals would be preferable. For example, we might be interested in doing away with the second `if` and reusing the resulting heterogeneous sub-circuit (shown in Figure 4) multiple times when building a more complex circuit, changing only the basis of the second measurement from time to time.

Giving a proper status to the programs that build this kind of heterogeneous circuits would thus endow the programming language with greater flexibility and modularity, so it is natural to wonder whether that of `Quipper` is an intrinsic limitation or if a richer type system can deal with a more general form of circuits.

In this paper, we introduce a generalization of Proto-Quipper-M, called Proto-Quipper-K, in which dynamic lifting is available in a very general form, even more so than in the original `Quipper` language. This newly introduced language is capable of producing not only circuits like the one in Figure 2, but rather a more general class of circuits whose structure and type *essentially depend* on the values flowing through the lifted channels, like the one in Figure 4. We show throughout the paper that this asks for a non-trivial generalization of Proto-Quipper-M's type system, in which types reflect the different behaviors a circuit can have. This is achieved through a type and effect system [16] which assigns any Proto-Quipper-K computation (possibly) distinct types depending on the state of the lifted variables. The main results, beside the introduction of the language itself, its type system, and its operational semantics, are the type soundness results of subject reduction and progress, which together let us conclude that well-typed Proto-Quipper-K programs do not go wrong. An extended version of this paper with more detailed definitions and proofs is available [2].

2 Circuits and Dynamic Lifting: a Bird's-Eye View

This section is meant to provide an informal introduction to the peculiarities of the Proto-Quipper family of paradigmatic programming languages, for the non-specialists. The host language, namely `Haskell`, is modeled as a linearly typed λ-calculus. Terms, in addition

$$\lambda q_{\mathsf{Qubit}}.\lambda a_{\mathsf{Qubit}}.\mathsf{let}\ (q,a) = \mathsf{apply}(\mathit{CNOT},(q,a))\ \mathsf{in}$$
$$\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \ \mathsf{apply}(\mathit{Meas2},(q,a))$$

Figure 5 A Proto-Quipper-M program describing the circuit shown in Figure 1. We assume that we have a constant boxed circuit $\mathit{CNOT}, H, \mathit{Meas2}$, etc. for every available primitive operation.

to manipulating ordinary data structures and computing (possibly) higher-order functions, are allowed to act on an underlying circuit, which we usually refer to as C. During program evaluation, C can be modified with the addition of wires, gates or entire sub-circuits. This is made possible through a dedicated operator called apply. But how can the programmer specify *where* in C these modifications have to be carried out? This is possible thanks to the presence of *labels*, that is, names that identify distinct output wires of C. These labels are ordinary terms which can be passed around, but have to be treated linearly. Among other things, they can be passed to apply, together with the specification of which gate or sub-circuit is to be appended to the underlying circuit C.

But is apply the only way of manipulating circuits? The answer is negative. Circuits, once built by means of a term, can be "boxed", potentially copied, and passed to other parts of the program, where they can be used, usually by appending them to multiple parts of the underlying circuit. From a linguistic point of view, this is possible thanks to an additional operator, called box, which is responsible for turning a circuit-building term M of type $!(T \multimap U)$ – the type of duplicable functions from label tuples to label tuples – into a term of type $\mathsf{Circ}(T,U)$ – the type of circuits. The term $\mathsf{box}\, M$ does not touch the underlying circuit, but rather evaluates M "behind the scenes", in a sandboxed environment, to obtain a standalone circuit which is then returned as a *boxed circuit*.

Measure as a Label-Lifting Operation

The above considerations are agnostic to the kind of circuits being built. In fact, any type of circuit-like structure can be produced in output by programs of the Proto-Quipper family of languages, provided that it can be interpreted as a morphism in an underlying symmetric monoidal category [20]. If we imagine that the produced structure is an *actual* quantum circuit, however, it is only natural to wonder whether all the examples of circuits that we talked about informally in the introduction can be captured by some instance of Proto-Quipper. Unsurprisingly, the program in Figure 1 is not at all problematic, and can be handled easily by all languages in the Proto-Quipper family (see Figure 5).

The program in Figure 2, on the other hand, can only be handled by Proto-Quipper-L (see Figure 6) and Proto-Quipper-Dyn. In Proto-Quipper-L, a measurement can return the *Boolean value* corresponding to the outcome of the measurement. When this happens, the ongoing computation is split into two branches: one in which the Boolean output is true and one in which it is false. This way, further circuit-altering operations down the line can depend on the classical information produced by the intermediate measurement. In Proto-Quipper-Dyn on the other hand, a bespoke operator dynlift allows to turn a term of type Bit into a term of type Bool, whose result can be used to a similar effect.

How about the program in Figure 3? Unfortunately, this case exceeds the expressiveness of both Proto-Quipper-L and Proto-Quipper-Dyn, in that it requires the distinct execution branches to yield values of different *types*, although only temporarily. The two languages, on the other hand, require all branches of a computation to share the same type, like in `Quipper`. This is where our contribution starts.

$$
\begin{aligned}
&\lambda b_{\mathsf{Qubit}}.\lambda q_{\mathsf{Qubit}}.\lambda a_{\mathsf{Qubit}}.\mathsf{let}\ (q,a) = (\mathsf{unbox}\ \mathit{CNOT})\,(q,a)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\mathsf{let}\ q = (\mathsf{unbox}\ H)\,q\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\mathsf{let}\ (u,s) = (\mathsf{unbox}\ \mathit{MeasLift2})\,(q,a)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\mathsf{let}\ b = \mathsf{if}\ s\ \mathsf{then}\ (\mathsf{unbox}\ X)\,b\ \mathsf{else}\ b\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\mathsf{let}\ b = \mathsf{if}\ u\ \mathsf{then}\ (\mathsf{unbox}\ Z)\,b\ \mathsf{else}\ b\ \mathsf{in}\ b
\end{aligned}
$$

Figure 6 A Proto-Quipper-L program describing the circuit shown in Figure 2. Informally, terms of the form $(\mathsf{unbox}\ H)\,q$ correspond to terms of the form $\mathsf{apply}(H,q)$ in Proto-Quipper-M.

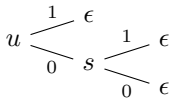


Figure 7 An example of a lifting tree. The empty tree is denoted by ϵ.

The Basic Ideas Underlying Proto-Quipper-K

The approach to dynamic lifting that we follow in this paper is radical. The evaluation of a term M involving the apply operator can give rise to the lifting of a bit value into a variable u and consequently produce in output not a single result in the set VAL of values, but possibly one distinct result for each possible value of u. Therefore, it is natural to think of M as a computation that results in an object in the set $\mathcal{K}_{\{u\}}(\mathit{VAL})$, where $\mathcal{K}_{\{u\}} = X \mapsto (\{u\} \to \{0,1\}) \to X$ is a functor such that, for each possible assignment of a Boolean value to u, $\mathcal{K}_{\{u\}}(X)$ returns an element of X.

What if *more than one* variable is lifted? For example, a program could lift s after having lifted u, but *only if* the latter has value 0. This shows that one cannot just take $\mathcal{K}_{\{u,s\}} = X \mapsto (\{u,s\} \to \{0,1\}) \to X$, simply because not all assignments in $\{u,s\} \to \{0,1\}$ are relevant. Instead, one should just focus on the three assignments $(u=0,s=0),(u=0,s=1)$ and $(u=1)$, namely those assignments which are consistent with the tree in Figure 7, which we call a *lifting tree*. This is a key concept in this work, which we will discuss in detail in Section 3. Our type system captures the lifting pattern of an underlying well-typed program through a lifting tree $\mathfrak{t}$ and the result of the corresponding computation is an element of $\mathcal{K}_{\mathfrak{t}}(\mathit{VAL})$ where $\mathcal{K}_{\mathfrak{t}} = X \mapsto (\mathscr{P}_{\mathfrak{t}} \to X)$ and $\mathscr{P}_{\mathfrak{t}}$ is the set of all assignments of variables which describe a root-to-leaf path in $\mathfrak{t}$. Since by design we want to handle situations in which a circuit, and by necessity the term building it, can produce results which have distinct *types* – and not only distinct *values* – depending on the values of the lifted variables, we also employ (in the spirit of the type and effects paradigm [16]) an effectful notion of *type*, in which computations are typed according to an element of $\mathcal{K}_{\mathfrak{t}}(\mathit{TYPE})$, where TYPE is the set of Proto-Quipper-K types.

3 Generalized Quantum Circuits

A quantum circuit describes a quantum computation by means of the application of *quantum gates*, which represent basic unitary operations, to a number of typed *input wires*, to obtain a number of typed *output wires*. In this section, we introduce a general form of circuits in which the application of *any* gate to one or more wires can be carried out *conditionally* on the classical value flowing in a lifted channel. This is possible even when the gate inputs are *not* the same number and type of the gate outputs. This implies that the number and type of the outputs of a circuit can depend on the values flowing in its wires.

M-types	*MTYPE*	T, U	$::= \mathbb{1} \mid w \mid T \otimes U.$
M-values	*MVAL*	$\vec{\ell}, \vec{k}$	$::= * \mid \ell \mid (\vec{\ell}, \vec{k}).$

$$\text{unit}\,\frac{}{\emptyset \vdash_m * : \mathbb{1}} \qquad \text{label}\,\frac{}{\ell : w \vdash_m \ell : w} \qquad \text{tuple}\,\frac{Q \vdash_m \vec{\ell} : T \quad L \vdash_m \vec{k} : U}{Q, L \vdash_m (\vec{\ell}, \vec{k}) : T \otimes U}$$

Figure 8 Syntax and rule system for M-types and M-values.

A Syntax for Circuits

We represent the inputs and outputs of a circuit as *label contexts*, that is, partial mappings from the set $\mathscr{L}$ of *label names* to the set $\mathscr{W} = \{\mathsf{Bit}, \mathsf{Qubit}\}$ of wire types. The set $\mathscr{L}$ contains precisely the kind of labels that we mentioned in Section 2, therefore a label context is a way to attach type information to labeled wires. We write the set of all label contexts as $\mathscr{Q}$.

Whereas the order of wires in a circuit as a whole is irrelevant, the order of wires in a *gate application* is crucial. For this reason, we perform gate applications not on label contexts, but rather on *label tuples*, which imbue label contexts with a specific ordering via a simple form of typing judgment. The grammar and typing rules for label tuples, which we call *M-values* and whose types we call *M-types*, following [20], are given in Figure 8. Note that $\ell \in \mathscr{L}, w \in \mathscr{W}$, and Q and L are label contexts whose disjoint union is denoted by Q, L.

▶ **Definition 1** (Gate Set). *Let $\mathscr{G}$ be a set of* gates*, equipped with two functions* $\mathsf{inType} : \mathscr{G} \to \mathit{MTYPE}$ *and* $\mathsf{outType} : \mathscr{G} \to \mathit{MTYPE}$*. We denote by $\mathscr{G}(T, U)$ the set of gates with input type T and output type U.*

As an example, $\mathscr{G}(\mathsf{Qubit}, \mathsf{Qubit})$ includes the so-called Hadamard gate, which is used to put a single qubit into a perfect superposition. Besides the set of labels $\mathscr{L}$, there is also another set of names, called $\mathscr{V}$, which is disjoint from it and contains the lifted variables. An *assignment* of lifted variables is then simply a finite sequence of equalities $(u_1 = p_1, \ldots, u_n = p_n)$ which assign the values $p_1, \ldots, p_n \in \{0, 1\}$ to the distinct variables $u_1, \ldots, u_n \in \mathscr{V}$, respectively. We usually indicate assignments with metavariables such as a, b and c.

We now introduce a low-level language to describe quantum circuits at the gate level, which will serve as a target for circuit building in Proto-Quipper-K. We call it *circuit representation language* (CRL) and define it via the following grammar:

$$C, D ::= \mathsf{input}(Q) \mid C; a \,?\, g(\vec{\ell}) \to \vec{k} \mid C; a \,?\, \mathsf{lift}(\ell) \Rightarrow u. \tag{1}$$

The base case $\mathsf{input}(Q)$ corresponds to the trivial identity circuit that takes as input the wires represented by Q and does nothing to them. The notation $a \,?\, g(\vec{\ell}) \to \vec{k}$ denotes the application of a gate g to the wires identified by $\vec{\ell}$ to obtain the wires in $\vec{k}$, provided that the condition expressed by a is met. We simply write $g(\vec{\ell}) \to \vec{k}$ when a gate is applied unconditionally (i.e. when $a = \emptyset$). Figure 9a shows a simple example of a CRL circuit consisting exclusively of gate applications.

On the other hand, $a \,?\, \mathsf{lift}(\ell) \Rightarrow u$ represents the dynamic lifting of the bit wire ℓ if the condition expressed by a is met. When we perform dynamic lifting on a bit, we promote its contents to a Boolean value that is bound to the lifted variable u. This variable can then be mentioned in subsequent assignments to control whether further operations in the circuit are executed or not. The introduction of u thus naturally leads to two distinct execution branches: one in which $u = 0$ and one in which $u = 1$. As we do for gate applications, we write $\mathsf{lift}(\ell) \Rightarrow u$ when a lifting operation is unconditional. A CRL circuit that performs dynamic lifting is shown in Figure 9b.

$\mathsf{input}(q_0 : \mathsf{Qubit}, a_0 : \mathsf{Qubit});$
$\mathsf{CNOT}(q_0, a_0) \to (q_1, a_1);$
$\mathsf{H}(q_1) \to q_2;$
$\mathsf{Meas2}(q_2, a_1) \to (x, y);$

(a) Circuit from Figure 1.

$\mathsf{input}(b_0 : \mathsf{Qubit}, q_0 : \mathsf{Qubit}, a_0 : \mathsf{Qubit});$
$\mathsf{CNOT}(q_0, a_0) \to (q_1, a_1);$
$\mathsf{H}(q_1) \to q_2;$
$\mathsf{Meas2}(q_2, a_1) \to (x, y);$
$\mathsf{lift}(x) \Rightarrow u;$
$\mathsf{lift}(y) \Rightarrow s;$
$(s = 1)\,?\,\mathsf{X}(b_0) \to b_1;$
$(u = 1; s = 0)\,?\,\mathsf{Z}(b_0) \to b_2;$
$(u = 1; s = 1)\,?\,\mathsf{Z}(b_1) \to b_3;$

(b) Circuit from Figure 2.

Figure 9 Two examples of CRL descriptions of a quantum circuit.

Lifting Trees

Naturally, not all CRL expressions denote reasonable circuits. For example, conditioning the application of a gate on the value of a lifted variable which has not yet been introduced should be avoided, for obvious reasons. Capturing this idea at the type level is nontrivial, since the presence of a variable can itself depend on previous liftings. This is where the concept of lifting tree, which we introduced informally in Section 2, really comes into play.

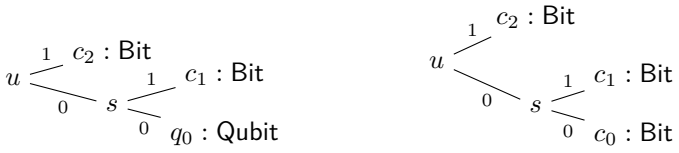
▶ **Definition 2** (Lifting Tree). *We define the set $\mathscr{T}$ of* lifting trees*, along with their* variable set on assignment *a, seen as a function $\mathsf{var}_a : \mathscr{T} \to \mathcal{P}(\mathscr{V})$, as the smallest set of expressions and functions such that*

- *$\epsilon \in \mathscr{T}$ with $\mathsf{var}_a(\epsilon) = \emptyset$.*
- *If $\mathsf{t}_0 \in \mathscr{T}$ and $\mathsf{t}_1 \in \mathscr{T}$, then for every u that is neither in $\mathsf{var}_\emptyset(\mathsf{t}_0)$ nor in $\mathsf{var}_\emptyset(\mathsf{t}_1)$ we have $u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\} \in \mathscr{T}$ and*

$$\mathsf{var}_a(u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\}) = \{u\} \cup \begin{cases} \mathsf{var}_a\,(\mathsf{t}_0) & \text{if } a(u) = 0, \\ \mathsf{var}_a\,(\mathsf{t}_1) & \text{if } a(u) = 1, \\ \mathsf{var}_a(\mathsf{t}_0) \cup \mathsf{var}_a(\mathsf{t}_1) & \text{if } a(u) \text{ is undefined.} \end{cases} \tag{2}$$

We often write $\mathsf{var}(\mathsf{t})$ as shorthand for $\mathsf{var}_\emptyset(\mathsf{t})$ to denote all the variables mentioned in t. By way of lifting trees, we can keep track of whether an assignment, representing a condition, is consistent with the current state of the lifted variables. Given a lifting tree t, we call $\mathscr{A}_\mathsf{t}$ the set of such assignments (which is easily defined by induction on t, see [2]). Among these consistent assignments, there are some which are *maximal*, i.e. that cannot be further extended: they describe root-to-leaf paths in t and correspond exactly to the elements of the set $\mathscr{P}_\mathsf{t}$ which we introduced in Section 2. Unsurprisingly, for all t we have $\mathscr{P}_\mathsf{t} \subseteq \mathscr{A}_\mathsf{t}$. As an example, let t be the tree from Figure 7. We have $\mathsf{var}(\mathsf{t}) = \{u, s\}, \mathscr{A}_\mathsf{t} = \{\emptyset, (u = 0), (u = 1), (s = 0), (s = 1), (u = 0, s = 0), (u = 0, s = 1)\}$ and $\mathscr{P}_\mathsf{t} = \{(u = 1), (u = 0, s = 0), (u = 0, s = 1)\}$.

Finally, we can formally define one of the key notions of this paper, not only for circuits, but also for programs: given a generic set X, $\mathcal{K}_\mathsf{t}(X)$ indicates the set $\mathscr{P}_\mathsf{t} \to X$, which we call the *lifting* of X, and whose elements we refer to as *lifted objects*. Despite the fact that



(a) Before composition. **(b)** After composition.

Figure 10 An example of composition with overwriting.

lifted objects are formally mappings, seeing them as decorated lifting trees whose leaves are labeled with objects from X is perhaps more intuitive. Following this interpretation, given $x \in X$, we indicate by $\{x\}$ the trivial lifted object in $\mathcal{K}_\epsilon(X)$ defined as the mapping $\emptyset \mapsto x$, and by $u\,\{\xi_0\}\{\xi_1\}$ the object in $\mathcal{K}_{u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\}}(X)$ defined as $a \mapsto \xi_{a(u)}(a')$, where $\xi_0 \in \mathcal{K}_{\mathsf{t}_0}(X), \xi_1 \in \mathcal{K}_{\mathsf{t}_1}(X)$ and $a' \in \mathscr{P}_{\mathsf{t}_{a(u)}}$ is obtained from a by excluding u from its domain. A graphical representation of this intuition can be found in the form of the trees shown in figures 10 and 11.

▶ **Example 3.** Consider Figure 9. The CRL circuit on the left does not perform lifting and therefore has a trivial output $\{x : \mathsf{Bit}, y : \mathsf{Bit}\}$, while the CRL circuit on the right does and has output $u\,\{s\,\{b_0 : \mathsf{Qubit}\}\{b_1 : \mathsf{Qubit}\}\}\{s\,\{b_2 : \mathsf{Qubit}\}\{b_3 : \mathsf{Qubit}\}\}$.

The same intuition informs the various operations that we define homogeneously on lifting trees and lifted objects. The first is a very natural one: if t is a lifting tree and $\{x_a\}_{a\in I}$ is a family of elements in X indexed on a subset $I \subseteq \mathscr{P}_\mathsf{t}$ of the root-to-leaf paths of t, then the expression $\mathsf{t}[x_a]^I_a$ stands for the lifted object obtained by sticking each x_a to the leaf of t identified by $a \in I$. Note that this operation, which we call *composition*, is generally loosely typed and can give rise to heterogeneous lifted objects. However, in the case $I = \mathscr{P}_\mathsf{t}$, the resulting lifted object belongs to $\mathcal{K}_\mathsf{t}(X)$. In this case we also write $\mathsf{t}[x_a]_a$ as shorthand for $\mathsf{t}[x_a]^{\mathscr{P}_\mathsf{t}}_a$, whereas when I is a singleton $\{b\}$ we usually omit the subscript a and write $\mathsf{t}[x]^{\{b\}}$ for $\mathsf{t}[x_a]^{\{b\}}_a$. We also allow already specified lifted objects to appear on the left of a composition, in which case we overwrite the interested leaves. For instance, if Δ is a lifted label context, we often write the expression $\Delta[Q]^{\{a\}}$ to denote the lifted label context that associates Q to a and is otherwise equal to Δ on all the other branches.

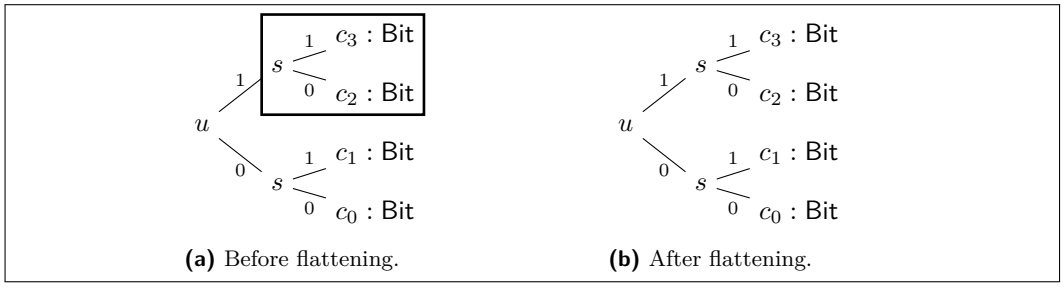
▶ **Definition 4** (Composition). *Given $\xi \in \mathcal{K}_\mathsf{t}(X)$, an index set $I \subseteq \mathscr{P}_\mathsf{t}$ and a family $\{x_a\}_{a\in I}$ of elements in X, we define the* composition of ξ and $\{x_a\}_{a\in I}$, *written $\xi[x_a]^I_a$, as*

$$\begin{aligned} \{y\}[x_a]^\emptyset_a &= \{y\}, \\ \{y\}[x_a]^{\{\emptyset\}}_a &= \{x_a\}, \\ u\,\{\xi_0\}\{\xi_1\}[x_a]^I_a &= u\,\{\xi_0[x_a]^{I_0}_a\}\{\xi_1[x_a]^{I_1}_a\}, \end{aligned} \tag{3}$$

where $I_b = \{a|_{\mathsf{var}(\mathsf{t})\setminus\{u\}} \mid a \in I \wedge a(u) = b\}$ and $a|_{\mathsf{var}(\mathsf{t})\setminus\{u\}}$ denotes the exclusion of u from the domain of a.

▶ **Example 5.** Let $\mathsf{t} = u\,\{s\,\{\epsilon\}\{\epsilon\}\}\{\epsilon\}$ and let $\Delta = u\,\{s\,\{q_0 : \mathsf{Qubit}\}\{c_1 : \mathsf{Bit}\}\}\{c_2 : \mathsf{Bit}\} \in \mathcal{K}_\mathsf{t}(\mathscr{Q})$ be a lifted label context, which graphically corresponds to the tree shown in Figure 10a. We have $\Delta[c_0 : \mathsf{Bit}]^{\{u=0,s=0\}} = u\,\{s\,\{c_0 : \mathsf{Bit}\}\{c_1 : \mathsf{Bit}\}\}\{c_2 : \mathsf{Bit}\} \in \mathcal{K}_\mathsf{t}(\mathscr{Q})$, which corresponds to the tree shown in Figure 10b.

The second operation is a *flattening* operation, which we indicate with $\lfloor\cdot\rfloor$. Intuitively, if we have a lifted object whose leaves are themselves trees, the flattening operation "unwraps" the trees in the leaves so that they become sub-trees of said lifted object. Given a family of



(a) Before flattening. **(b)** After flattening.

Figure 11 An example of flattening.

trees $\{\mathfrak{r}_a\}_{a\in\mathscr{P}_\mathfrak{t}}$ the difference between $\mathfrak{t}[\mathfrak{r}_a]_a$ and $\lfloor \mathfrak{t}[\mathfrak{r}_a]_a \rfloor$ is therefore that the former is an element of $\mathcal{K}_\mathfrak{t}(\mathscr{T})$, whereas the latter is a proper element of $\mathscr{T}$. This operation is well-defined when $\mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}_a) = \emptyset$ for every $a \in \mathscr{P}_\mathfrak{t}$. We can also flatten when we have have lifted objects on the right side of a composition: if we have $\mathfrak{t}[\xi_a]_a$, where $\xi_a \in \mathcal{K}_{\mathfrak{r}_a}(X)$ for every $a \in \mathscr{P}_\mathfrak{t}$, then $\lfloor \mathfrak{t}[\xi_a]_a \rfloor \in \mathcal{K}_{\lfloor \mathfrak{t}[\mathfrak{r}_a]_a \rfloor}(X)$.

A formal definition of flattening requires us to consider a slightly more general operation $\lfloor\cdot\rfloor^{\mathcal{V}}$, where $\mathcal{V}$ is a finite set of lifted variables which are accumulated as a lifted object is traversed. At the leaf level, $\lfloor\{x\}\rfloor^{\mathcal{V}} = x$ only if x is a lifted object in which none of the lifted variables in $\mathcal{V}$ occur (a condition expressed in the first line of Equation 4). In case of name clashes, the whole operation is undefined and a renaming of lifted variables is required prior to composition and flattening.

▶ **Definition 6** (Flattening). *Given $\xi \in \mathcal{K}_\mathfrak{t}(X)$, we define the* flattening of ξ under $\mathcal{V}$, *written* $\lfloor\xi\rfloor^{\mathcal{V}}$, *as*

$$\lfloor\{y\}\rfloor^{\mathcal{V}} = \begin{cases} y & \text{if } \exists \mathfrak{r}, Y \text{ s.t. } y \in \mathcal{K}_\mathfrak{r}(Y) \text{ and } \mathcal{V} \cap \mathsf{var}(\mathfrak{r}) = \emptyset, \\ \{y\} & \text{if } \nexists \mathfrak{r}, Y \text{ s.t. } y \in \mathcal{K}_\mathfrak{r}(Y), \end{cases} \tag{4}$$
$$\lfloor u\,\{\xi_0\}\{\xi_1\}\rfloor^{\mathcal{V}} = u\,\{\lfloor\xi_0\rfloor^{\mathcal{V}\cup\{u\}}\}\{\lfloor\xi_1\rfloor^{\mathcal{V}\cup\{u\}}\}.$$

The actual flattening operation that we employ in the rest of the paper can then be defined as $\lfloor\cdot\rfloor = \lfloor\cdot\rfloor^{\emptyset}$.

▶ **Example 7.** Reconsider $\mathfrak{t}$ and $u\,\{s\,\{c_0 : \mathsf{Bit}\}\{c_1 : \mathsf{Bit}\}\}\{c_2 : \mathsf{Bit}\} = \Delta'$ from Example 5. Let $\xi = \Delta'[s\,\{c_2 : \mathsf{Bit}\}\{c_3 : \mathsf{Bit}\}]^{\{u=1\}} = u\,\{s\,\{c_0 : \mathsf{Bit}\}\{c_1 : \mathsf{Bit}\}\}\{\{s\,\{c_2 : \mathsf{Bit}\}\{c_3 : \mathsf{Bit}\}\}\} \in \mathcal{K}_\mathfrak{t}(\mathscr{Q} \cup \mathcal{K}_{s\,\{\epsilon\}\{\epsilon\}}(\mathscr{Q}))$. Note that this object, corresponding to Figure 11a, is *not* properly a lifted label context, as one of its leaves is itself a lifted label context. Because s does not occur in $\mathsf{var}_{u=1}(\mathfrak{t}) = \{u\}$, we can write $\lfloor\xi\rfloor = u\,\{s\,\{c_0 : \mathsf{Bit}\}\{c_1 : \mathsf{Bit}\}\}\{s\,\{c_2 : \mathsf{Bit}\}\{c_3 : \mathsf{Bit}\}\} \in \mathcal{K}_{\mathfrak{t}'}(\mathscr{Q})$, for $\mathfrak{t}' = u\,\{s\,\{\epsilon\}\{\epsilon\}\}\{s\,\{\epsilon\}\{\epsilon\}\}$. *This* is an actual lifted label context, which corresponds to Figure 11b.

In conjunction, these two operations greatly simplify our treatment of dynamic lifting, as they allow us to describe in detail the desired shape of a lifted object. For example, in later sections we often write $\Delta = \lfloor\Delta'[\Delta'', \Lambda]^{\{a\}}\rfloor$ to say that the lifted label context Δ is such that if we start from its root and follow the path described by a, we find a sub-tree Δ'', Λ (the disjoint union is lifted in a natural way, as the disjoint union of the corresponding leaves of Δ'' and Λ when these have the same underlying lifted tree), and that we are interested in only *some* elements of the corresponding lifted label context, for instance those that occur in Λ.

$$\mathrm{id}\,\frac{}{\mathsf{input}(Q) \vdash^{\epsilon} Q \triangleright \{Q\}} \qquad \mathrm{lift}\,\frac{C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} \ell : \mathsf{Bit} \quad a \in \mathscr{A}_{\mathfrak{t}} \quad u \notin \mathsf{var}_a(\mathfrak{t})}{C; a\,?\,\mathsf{lift}(\ell) \Rightarrow u \vdash^{\mathfrak{t} \prec^a u\,\{\epsilon\}\{\epsilon\}} Q \triangleright \Delta \prec^a u\,\{\epsilon\}\{\epsilon\}}$$

$$\mathrm{gate}\,\frac{C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} Q' \quad a \in \mathscr{A}_{\mathfrak{t}} \quad g \in \mathscr{G}(T,U) \quad Q' \vdash_m \vec{\ell} : T \quad L \vdash_m \vec{k} : U \quad \mathsf{fresh}(\vec{k}, C)}{C; a\,?\,g(\vec{\ell}) \to \vec{k} \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} L}$$

Figure 12 The rules for CRL circuit signatures.

A Formal System for Circuit Signatures

Now that we have introduced lifting trees and lifted objects as a means to reason about dynamic lifting, we are ready to introduce the notion of *signature* of a circuit.

▶ **Definition 8** (Circuit Signature). *Given a circuit C, a lifting tree $\mathfrak{t}$, a label context Q and a lifted label context Δ, a* circuit signature *is an expression of the form $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$.*

Informally, $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$ means that C takes as input the labels in Q, performs lifting according to tree $\mathfrak{t}$ and outputs any of the leaves in Δ, which is a lifted label context backed by $\mathfrak{t}$. More formally, a valid circuit signature is derived by the rules in Figure 12. Note that $\Delta \overset{a}{;} Q'$ represents the extension of Δ with Q' on all leaves reachable by an assignment consistent with a. Formally, $\Delta \overset{a}{;} Q'$ is shorthand for $\Delta'[Q', Q_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$ if $\Delta = \Delta'[Q_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$, where $\mathscr{P}_{\mathfrak{t}}^a$ contains the paths in $\mathscr{P}_{\mathfrak{t}}$ which extend $a \in \mathscr{A}_{\mathfrak{t}}$. On the other hand, for any $\xi \in \mathcal{K}_{\mathfrak{t}}(X)$ and $\mathfrak{r} \in \mathscr{T}$ such that $\mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset$, we write $\xi \prec^a \mathfrak{r}$ to denote ξ in which every leaf x reachable by an assignment consistent with a is expanded to a sub-tree $\mathfrak{r}$ whose leaves are all x. More formally, $\xi \prec^a \mathfrak{r}$ is shorthand for $\lfloor \xi'[\mathfrak{r}[x_b]_c]_b^{\mathscr{P}_{\mathfrak{t}}^a} \rfloor$, if $\xi = \xi'[x_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$.

4 Proto-Quipper-K

We are finally ready to introduce Proto-Quipper-K, a programming language designed exactly to manipulate the kind of circuits that we presented in the previous section and guarantee the degree of flexibility that we mentioned in Section 2.

4.1 Types and Terms

The types and syntax of Proto-Quipper-K are given in Figure 13, where x and y range over the set of variable names, $u_1, \ldots, u_n$ over the set $\mathscr{V}$ of lifted variable names, $\mathfrak{t}$ over the set $\mathscr{T}$ of lifting trees and Greek letters generally indicate lifted objects. More precisely, $\alpha, \beta \in \mathcal{K}_{\mathfrak{t}}(TYPE), \upsilon \in \mathcal{K}_{\mathfrak{t}}(MTYPE), \mu \in \mathcal{K}_{\mathfrak{t}}(TERM)$ and $\lambda \in \mathcal{K}_{\mathfrak{t}}(MVAL)$, each for some $\mathfrak{t}$. Note that parameter types are the types given to non-linear resources, i.e. duplicable values. Note also that the M-values and M-types that we introduced in Section 3 are now a proper subset of Proto-Quipper-K's values and types, respectively.

A value of the form $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ is what we called a *boxed circuit* in Section 2, that is, a datum representation of a circuit C that takes as input the labels in the tuple $\vec{\ell}$, performs lifting according to $\mathfrak{t}$ and outputs one of the possible label tuples of λ depending on the lifted variables in $\mathfrak{t}$. Correspondingly, parameter types of the form $\mathsf{Circ}_{\mathfrak{t}}(T, \upsilon)$ are called *circuit types* and represent boxed circuits. Both boxed circuits and circuit types abstract over the lifted variables in $\mathsf{var}(\mathfrak{t})$ and thus enjoy a notion of α-equivalence. The box and apply constructs are those described in Section 2 and they respectively introduce and consume boxed circuits. Specifically, the programmer is never expected to write values of the form $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ by hand.

Types	$TYPE$	A, B	$::= \mathbb{1} \mid w \mid A \multimap_{\mathfrak{t}} \beta \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, U) \mid A \otimes B.$
Parameter Types	$PTYPE$	P, R	$::= \mathbb{1} \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, U) \mid P \otimes R.$
M-types	$MTYPE$	T, U	$::= \mathbb{1} \mid w \mid T \otimes U.$
Terms	$TERM$	M, N	$::= VW \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu \mid \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M$ $\mid \mathsf{force}\, V \mid \mathsf{box}_T\, V \mid \mathsf{apply}_{u_1,\dots,u_n}(V, W) \mid \mathsf{return}\, V.$
Values	VAL	V, W	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\vec{\ell}, C, \lambda)_{\mathfrak{t}} \mid (V, W).$
M-values	$MVAL$	$\vec{\ell}, \vec{k}$	$::= * \mid \ell \mid (\vec{\ell}, \vec{k}).$

Figure 13 Types and terms of Proto-Quipper-K.

Rather, they are expected to introduce the desired circuit by supplying an appropriate circuit-building term M to the box operator, obtaining a term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$. The use of lift guarantees that M does not make use of any linear resources from the current environment, i.e. that it can be safely evaluated in an isolated environment to produce C. After a boxed circuit $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ is introduced, the programmer can potentially copy it and apply it to the underlying circuit D via a term of the form $\mathsf{apply}_{u_1,\dots,u_n}((\vec{\ell}, C, \lambda)_{\mathfrak{t}}, \vec{k})$. Such a term "unboxes" C, finds the wires identified by $\vec{k}$ among the outputs of D and appends C to them. In this process, any lifted variables in $\mathsf{var}(\mathfrak{t})$, which were abstracted in $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$, have to be instantiated with concrete names. To this end, the programmer supplies the $n = |\mathsf{var}(\mathfrak{t})|$ lifted variables $u_1, \dots, u_n$, which are expected to be fresh.

Notice that there exists a strong distinction between values and terms, the latter representing effectful computations that can introduce new lifted variables as a consequence of dynamic lifting. This choice does not detract from the expressiveness of the language, since first and foremost a value V can always be turned into an effectless computation $\mathsf{return}\, V$. Furthermore, we have that terms such as MN can still be recovered in Proto-Quipper-K as $\mathsf{let}\ x = M\ \mathsf{in}\ \{\mathsf{let}\ y = N\ \mathsf{in}\ \{xy\}\}$. The let construct is in fact a central construct in Proto-Quipper-K: on top of serving as a sequencing operator it also doubles as a conditional statement. When we evaluate the term $\mathsf{let}\ x = M\ \mathsf{in}\ \mu$, we first carry out the computation described by M, which performs dynamic lifting according to some lifting tree $\mathfrak{t}$ and consequently results in a lifted value $\phi \in \mathcal{K}_{\mathfrak{t}}(VAL)$. At this point, we are not limited to passing each and every possible value of ϕ to the *same* continuation. Rather, we can define a different continuation for every possible outcome of the liftings in $\mathfrak{t}$. To this effect, the programmer supplies a lifted term $\mu \in \mathcal{K}_{\mathfrak{t}}(TERM)$, which matches ϕ's lifting tree and thus effectively provides such a roster of continuations. The following example is meant to help convey the role of let as a control flow operator.

▶ **Example 9.** Imagine we wanted to measure qubit ℓ, dynamically lift its value into a variable u and then apply the Hadamard gate to a second qubit k *only if* $u = 1$. Suppose we had CRL definitions $ML = \mathsf{input}(\ell : \mathsf{Qubit}); \mathsf{Meas}(\ell) \to \ell'; \mathsf{lift}(\ell') \Rightarrow u$ and $H = \mathsf{input}(\ell : \mathsf{Qubit}); \mathsf{H}(\ell) \to \ell'$, corresponding to the circuit that measures and then lifts a qubit and the circuit that just applies the Hadamard gate to its input, respectively. We would write the following Proto-Quipper-K program:

$$\begin{aligned} &\mathsf{let}\ \_ = \mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, \ell)\ \mathsf{in} \\ &\quad u\,\{\mathsf{return}\, k\}\{\mathsf{apply}((\ell, H, \{\ell'\})_\epsilon, k)\}, \end{aligned} \tag{5}$$

which may appear more familiar under the following `Haskell`-like syntactic sugar:

$$\begin{aligned} &\mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, \ell) \\ &\mathsf{when}\ (u = 1)\ \ \mathsf{apply}((\ell, H, \{\ell'\})_\epsilon, k). \end{aligned} \tag{6}$$

⌟

$$\lambda q_{\mathsf{Qubit}}.\,\mathsf{return}\,\lambda a_{\mathsf{Qubit}}.\mathsf{let}\,q = \mathsf{apply}((\ell, H, \{\ell'\}), q)\,\mathsf{in}\,\{ \\ \mathsf{let}\,\_ = \mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, q)\,\mathsf{in} \\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}((\ell, Meas_\alpha, \{\ell'\}), a)\}\}$$

Figure 14 A Proto-Quipper-K program describing the circuit shown in Figure 4.

In light of this example, we can see how the circuit shown in Figure 4 can be described in Proto-Quipper-K through a program such as the one in Figure 14. To conclude this section, recall that we mentioned earlier that whenever we apply a boxed circuit we need to instantiate its lifted variables with concrete names. This process is formalized through the lifted variable analog of substitution, which we call *renaming*.

▶ **Definition 10** (Renaming of Lifted Variables). *Given a lifted variable-bearing object x and a permutation π of $\mathscr{V}$, we call π a* renaming of lifted variables *and we define $x\langle\pi\rangle$ as x in which every occurrence of a lifted variable u is replaced by $\pi(u)$. We denote by $s_1/u_1, \ldots, s_n/u_n$ a permutation that exchanges s_1 for $u_1, \ldots s_n$ for u_n.*

4.2 Proto-Quipper-K's Typing Rules

At its core, Proto-Quipper-K's type system is a linear type system which distinguishes between *computations* (i.e. terms), which can be effectful, and *values*. We therefore introduce two kinds of typing judgments. One for terms, which are given lifted types, and one for values, which have regular types.

▶ **Definition 11** (Typing Judgments). *Given a typing context Γ, a label context Q, a term M and a lifted type α, a* computational typing judgment *is an expression of the form*

$$\Gamma; Q \vdash_c^{\mathfrak{t}} M : \alpha. \tag{7}$$

Given Γ, Q, a value V and a type A, a value typing judgment *is an expression of the form*

$$\Gamma; Q \vdash_v V : A. \tag{8}$$

When a typing context contains *exclusively* variables with parameter types, we write it as Φ, but in general a typing context Γ can contain both linear and parameter variables. Typing judgments are derived via the rules in Figure 15, where $s_1, \ldots, s_n$ range over $\mathscr{V}$. Note that we assume that $\mathscr{V}$ is totally ordered, so that when we write $\mathsf{var}(\mathfrak{t}) = \{u_1, \ldots, u_n\}$, e.g. in the *apply* rule, we have $u_1 \leq u_2 \leq \cdots \leq u_n$. The relational symbols $\Vdash_c$ and $\Vdash_v$ denote the lifting of the computation and value typing judgments to some tree $\mathfrak{t}$.

▶ **Definition 12** (Lifted Typing Judgments). *Given a lifting tree $\mathfrak{t}$, if for all $a \in \mathscr{P}_{\mathfrak{t}}$ we have $\Gamma_a; Q_a \vdash_c^{\mathfrak{r}_a} M_a : \alpha_a$, then we write*

$$\mathfrak{t}[\Gamma_a]_a; \mathfrak{t}[Q_a]_a \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mathfrak{t}[M_a]_a : \mathfrak{t}[\alpha_a]_a. \tag{9}$$

If for all $a \in \mathscr{P}_{\mathfrak{t}}$ we have $\Gamma_a; Q_a \vdash_v V_a : A_a$, then we write

$$\mathfrak{t}[\Gamma_a]_a; \mathfrak{t}[Q_a]_a \Vdash_v^{\mathfrak{t}} \mathfrak{t}[V_a]_a : \mathfrak{t}[A_a]_a. \tag{10}$$

For convenience, within such judgments, every lifted object (e.g. Δ in $\emptyset; \Delta \Vdash_v^{\mathfrak{t}} \lambda : \upsilon$ in the *circ* rule) is assumed to be backed by $\mathfrak{t}$, whereas every component which is *not* a lifted object (e.g. Φ or x in $\Phi, \Gamma_2, x : \alpha; Q_2 \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mu : \theta$ in the *let* rule) is assumed to be constant across the branches of $\mathfrak{t}$.

$$\text{unit}\frac{}{\Phi;\emptyset \vdash_v * : \mathbb{1}} \qquad \text{var}\frac{}{\Phi, x : A; \emptyset \vdash_v x : A} \qquad \text{label}\frac{}{\Phi; \ell : w \vdash_v \ell : w}$$

$$\text{abs}\frac{\Gamma, x : A; Q \vdash_c^{\mathsf{t}} M : \beta}{\Gamma; Q \vdash_v \lambda x_A.M : A \multimap_{\mathsf{t}} \beta} \qquad \text{app}\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \multimap_{\mathsf{t}} \beta \quad \Phi, \Gamma_2; Q_2 \vdash_v W : A}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathsf{t}} VW : \beta}$$

$$\text{let}\frac{\Phi, \Gamma_1; Q_1 \vdash_c^{\mathsf{t}} M : \alpha \quad \mu \in \mathcal{K}_{\mathsf{t}}(TERM) \quad \Phi, \Gamma_2, x : \alpha; Q_2 \Vdash_c^{\mathsf{t}[\mathfrak{r}_a]_a} \mu : \theta}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\lfloor \mathsf{t}[\mathfrak{r}_a]_a \rfloor} \mathsf{let}\ x = M\ \mathsf{in}\ \mu : \lfloor \theta \rfloor}$$

$$\text{tuple}\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \quad \Phi, \Gamma_2; Q_2 \vdash_v W : B}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_v (V, W) : A \otimes B}$$

$$\text{dest}\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \otimes B \quad \Phi, \Gamma_2, x : A, y : B; Q_2 \vdash_c^{\mathsf{t}} M : \alpha}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathsf{t}} \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M : \alpha} \qquad \text{lift}\frac{\Phi; \emptyset \vdash_c^{\epsilon} M : \alpha}{\Phi; \emptyset \vdash_v \mathsf{lift}\, M : !\alpha}$$

$$\text{force}\frac{\Gamma; Q \vdash_v V : !\alpha}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{force}\, V : \alpha} \qquad \text{box}\frac{\Gamma; Q \vdash_v V : !\{T \multimap_{\mathsf{t}} \upsilon\}}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{box}_T\, V : \{\mathsf{Circ}_{\mathsf{t}}(T, \upsilon)\}}$$

$$\text{apply}\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : \mathsf{Circ}_{\mathsf{t}}(T, \upsilon) \quad \Phi, \Gamma_2; Q_2 \vdash_v W : T \quad \mathsf{var}(\mathsf{t}) = \{u_1, \ldots, u_n\} \quad \pi = s_1/u_1, \ldots, s_n/u_n}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathsf{t}\langle\pi\rangle} \mathsf{apply}_{s_1,\ldots,s_n}(V, W) : \upsilon\langle\pi\rangle}$$

$$\text{circ}\frac{C \vdash^{\mathsf{t}} Q \triangleright \Delta \quad \emptyset; Q \vdash_v \vec{\ell} : T \quad \emptyset; \Delta \Vdash_v^{\mathsf{t}} \lambda : \upsilon}{\Phi; \emptyset \vdash_v (\vec{\ell}, C, \lambda)_{\mathsf{t}} : \mathsf{Circ}_{\mathsf{t}}(T, \upsilon)} \qquad \text{return}\frac{\Gamma; Q \vdash_v V : A}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{return}\, V : \{A\}}$$

Figure 15 The typing rules of Proto-Quipper-K.

The *let* rule is unsurprisingly the most interesting rule of the system: if M is a term of lifted type $\alpha \in \mathcal{K}_{\mathsf{t}}(TYPE)$, μ is a lifted term with the same underlying tree structure t and for every root-to-leaf path a in t we have $\Phi, \Gamma_2, x : \alpha; Q_2 \vdash_c^{\mathfrak{r}_a} \mu(a) : \theta(a)$ (note that this last condition is captured by the lifted typing judgment), then the lifted type of $\mathsf{let}\ x = M\ \mathsf{in}\ \mu$ is obtained by simply flattening the generic lifted object θ to $\lfloor\theta\rfloor \in \mathcal{K}_{\lfloor \mathsf{t}[\mathfrak{r}_a]_a \rfloor}(TYPE)$. The *apply* rule also plays a pivotal role, as it actually introduces lifting into the type of a term. If V is a value of type $\mathsf{Circ}_{\mathsf{t}}(T, \upsilon)$, and thus corresponds to a boxed circuit which performs lifting according to t, then applying V to some appropriate wires W of the underlying circuit introduces the same lifting pattern into the computation, which has type $\upsilon\langle\pi\rangle \in \mathcal{K}_{\mathsf{t}\langle\pi\rangle}(MTYPE)$. The renaming of lifted variables π is required to avoid name clashes.

An example of a type derivation that leverages the full expressiveness of Proto-Quipper-K's type system is the one for the program shown in Figure 14 (describing the circuit in Figure 4), which can be found in Appendix A. To conclude this section, notice how just like M-values and M-types are a subset of the values and types of Proto-Quipper-K, the type system for M-values is in a one-to-one correspondence with a subset of Proto-Quipper-K's type system.

4.3 A Big-Step Operational Semantics

The big-step operational semantics of the language is based on an evaluation relation $(C, a, M) \Downarrow (D, \phi)$, where ϕ is a lifted value. This means that the term M evaluates to one of the possible values in ϕ, depending on the outcome of intermediate measurements, and updates the underlying circuit C upon branch a as a side effect, obtaining an updated circuit D. We call (C, a, M) a *left configuration* and (D, ϕ) a *right configuration*. Before we give the actual rules of the semantics, we must first give some definitions.

First and foremost, note that we are not as interested in the actual names of labels within a boxed circuit as much as we are in the structure that they convey. In fact, when we apply circuits to one another, it might be necessary to rename some of the labels occurring in the applicand in order to avoid naming conflicts, all the while preserving its structure. For this reason, whenever two circuits share the same structure and only differ by their respective labels, we consider them to be equivalent.

▶ **Definition 13** (Equivalent Boxed Circuits). *We say that two boxed circuits* $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ *and* $(\vec{\ell}', C', \lambda')_{\mathfrak{t}}$ *are* equivalent*, and we write* $(\vec{\ell}, C, \lambda)_{\mathfrak{t}} \cong (\vec{\ell}', C', \lambda')_{\mathfrak{t}}$*, when they only differ by a renaming of labels.*

Next, we define the two operations that actually implement the semantics of apply. The circuit insertion function $::_a$ is just a simple concatenation function defined on CRL circuits and all the heavy lifting is actually performed by the append function. This function is responsible for the actual unboxing of a boxed circuit, the renaming of its labels (to match the outputs of the underlying circuit and avoid labeling conflicts), the instantiation of the abstracted lifted variable names within it and its insertion in the underlying circuit.

▶ **Definition 14** (Insertion of Circuits). *Suppose* C *and* D *are two circuits. We define* the insertion of D in C on branch a*, written* $C ::_a D$ *as:*

$$\begin{aligned} C ::_a \mathsf{input}(Q) &= C, \\ C ::_a (D'; b \,?\, g(\vec{\ell}) \to \vec{k}) &= (C ::_a D'); a \cup b \,?\, g(\vec{\ell}) \to \vec{k}, \\ C ::_a (D'; b \,?\, \mathsf{lift}(\ell) \Rightarrow u) &= (C ::_a D'); a \cup b \,?\, \mathsf{lift}(\ell) \Rightarrow u, \end{aligned} \tag{11}$$

where $a \cup b$ *denotes the union of two assignments with disjoint domains.*

▶ **Definition 15** (append). *Suppose* $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$ *is a circuit and* $(\vec{\ell}, D, \lambda)_{\mathfrak{r}}$ *is a boxed circuit with* $\mathsf{var}(\mathfrak{r}) = \{u_1, \ldots, u_n\}$*. Suppose* $a \in \mathscr{P}_{\mathfrak{t}}$ *and let* $\vec{k}$ *be a label tuple whose labels all occur in* $\Delta(a)$*. Finally, let* $s_1, \ldots, s_n$ *be a sequence of distinct lifted variable names which do not occur in* $\mathsf{var}_a(\mathfrak{t})$*. We define* $\mathsf{append}(C, a, \vec{k}, (\vec{\ell}, D, \lambda)_{\mathfrak{r}}, s_1, \ldots, s_n)$ *as the function that*

1. *Finds* $(\vec{k}, D', \lambda')_{\mathfrak{t}} \cong (\vec{\ell}, D, \lambda)_{\mathfrak{t}}$ *such that all the labels occurring in* D'*, but not in* $\vec{k}$*, are fresh in* C*,*
2. *Computes* $D'' = D'\langle s_1/u_1, \ldots, s_n/u_n\rangle$ *and* $\lambda'' = \lambda'\langle s_1/u_1, \ldots, s_n/u_n\rangle$*,*
3. *Returns* $(C ::_a D'', \lambda'')$*.*

The rules of the operational semantics can be found in Figure 16, where $\mathsf{freshlabels}(T)$ produces a pair $(Q, \vec{\ell})$ such that $Q \vdash_m \vec{\ell} : T$. Note that, for each $\mathfrak{t}$, we assume to have a total order over the elements of $\mathscr{P}_{\mathfrak{t}}$, so that when we write $\mathscr{P}_{\mathfrak{t}} = \{a_1, \ldots, a_n\}$ we have $a_1 \leq a_2 \leq \cdots \leq a_n$ and the semantics is deterministic. In order to prove progress in Section 5, we also consider a notion of divergence for configurations. Intuitively, a configuration (C, a, M) *diverges*, and we write $(C, a, M) \Uparrow$, when its evaluation does not terminate. More formally, divergence is defined coinductively by means of the rules in Figure 17.

5 Type Soundness

In general, the well-typedness of a Proto-Quipper-K configuration strongly depends on the underlying circuit. Specifically, a term M is well-typed when all the free labels occurring in it can be found with the appropriate type in the outputs of the underlying circuit, on the branch that M is manipulating. For this reason, we give the following notions of well-typedness.

$$\text{app}\frac{(C, a, M[V/x]) \Downarrow (D, \phi)}{(C, a, (\lambda x_A.M)V) \Downarrow (D, \phi)} \qquad \text{dest}\frac{(C, a, M[V/x, W/y]) \Downarrow (D, \phi)}{(C, a, \mathsf{let}\ (x, y) = (V, W)\ \mathsf{in}\ M) \Downarrow (D, \phi)}$$

$$\text{let}\frac{\begin{array}{c}(C, a, M) \Downarrow (C_1, \phi) \qquad \phi \in \mathcal{K}_{\mathfrak{t}}(VAL) \qquad \mu \in \mathcal{K}_{\mathfrak{t}}(TERM) \\ \mathscr{P}_{\mathfrak{t}} = \{a_1, \ldots, a_n\} \quad (C_i, a \cup a_i, \mu(a_i)[\phi(a_i)/x]) \Downarrow (C_{i+1}, \psi_{a_i}) \ \text{ for } \ i = 1, \ldots, n\end{array}}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Downarrow (C_{n+1}, \lfloor \mathfrak{t}[\psi_a]_a \rfloor)}$$

$$\text{force}\frac{(C, a, M) \Downarrow (D, \phi)}{(C, a, \mathsf{force}(\mathsf{lift}\ M)) \Downarrow (D, \phi)} \qquad \text{apply}\frac{(C', \lambda') = \mathsf{append}(C, a, \vec{k}, (\vec{\ell}, D, \lambda)_{\mathfrak{t}}, s_1, \ldots, s_n)}{(C, a, \mathsf{apply}_{s_1,\ldots,s_n}((\vec{\ell}, D, \lambda)_{\mathfrak{t}}, \vec{k})) \Downarrow (C', \lambda')}$$

$$\text{box}\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(T) \quad (\mathsf{input}(Q), \emptyset, \mathsf{let}\ x = M\ \mathsf{in}\ \{x\vec{\ell}\}) \Downarrow (D, \lambda) \quad \lambda \in \mathcal{K}_{\mathfrak{t}}(MVAL)}{(C, a, \mathsf{box}_T(\mathsf{lift}\ M)) \Downarrow (C, \{(\vec{\ell}, D, \lambda)_{\mathfrak{t}}\})}$$

$$\text{return}\frac{}{(C, a, \mathsf{return}\ V) \Downarrow (C, \{V\})}$$

Figure 16 The big-step operational semantics of Proto-Quipper-K.

$$\text{app}\frac{(C, a, M[V/x]) \Uparrow}{(C, a, (\lambda x_A.M)V) \Uparrow} \qquad \text{dest}\frac{(C, a, M[V/x, W/y]) \Uparrow}{(C, a, \mathsf{let}\ (x, y) = (V, W)\ \mathsf{in}\ M) \Uparrow}$$

$$\text{force}\frac{(C, a, M) \Uparrow}{(C, a, \mathsf{force}(\mathsf{lift}\ M)) \Uparrow} \qquad \text{let-now}\frac{(C, a, M) \Uparrow}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Uparrow}$$

$$\text{let-then}\frac{\begin{array}{c}(C, a, M) \Downarrow (C_1, \phi) \qquad \phi \in \mathcal{K}_{\mathfrak{t}}(VAL) \qquad \mu \in \mathcal{K}_{\mathfrak{t}}(TERM) \\ \mathscr{P}_{\mathfrak{t}} = \{a_1, \ldots, a_n\} \quad (C_i, a \cup a_i, \mu(a_i)[\phi(a_i)/x]) \Downarrow (C_{i+1}, \psi_{a_i}) \ \text{ for } \ i = 1, \ldots, j-1 \\ (C_j, a \cup a_j, \mu(a_j)[\phi(a_j)/x]) \Uparrow\end{array}}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Uparrow}$$

$$\text{box}\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(T) \quad (\mathsf{input}(Q), \emptyset, \mathsf{let}\ x = M\ \mathsf{in}\ \{x\vec{\ell}\}) \Uparrow}{(C, a, \mathsf{box}_T(\mathsf{lift}\ M)) \Uparrow}$$

Figure 17 The big-step divergence rules of Proto-Quipper-K.

▶ **Definition 16** (Well-Typed Configuration). *We say that*

- *a left configuration* (C, a, M) *is* well-typed with input Q, past lifting tree $\mathfrak{t}$, future lifting tree $\mathfrak{r}$, lifted type α and outputs Δ, *and we write* $Q \vdash^{\mathfrak{r}}_{\mathfrak{t}} (C, a, M) : \alpha; \Delta$, *when* $a \in \mathscr{P}_{\mathfrak{t}}, \mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset, C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} Q'$ *and* $\emptyset; Q' \vdash^{\mathfrak{r}}_{c} M : \alpha$,
- *a right configuration* (C, ϕ) *is* well-typed in the a branch with input Q, overall lifting tree $\mathfrak{t}$, lifted type α and outputs Δ, *and we write* $Q \vdash^{a}_{\mathfrak{t}} (C, \phi) : \alpha; \Delta$, *when* $\mathfrak{t} = \lfloor \mathfrak{t}'[\mathfrak{r}]^{\{a\}} \rfloor, \Delta = \lfloor \Delta'[\Delta'']^{\{a\}} \rfloor, C \vdash^{\mathfrak{t}} Q \triangleright \lfloor \Delta'[\Delta'', \Lambda]^{\{a\}} \rfloor$ *and* $\emptyset; \Lambda \Vdash^{\mathfrak{r}}_{v} \phi : \alpha$.

That being said, we are mainly interested in *closed* computations, in which the evaluation of a term builds the underlying circuit entirely from scratch. That is, we are interested in computations that start from configurations of the form $(\mathsf{input}(\emptyset), \emptyset, M)$, for some M. Whenever $(\mathsf{input}(\emptyset), \emptyset, M) \Downarrow (C, \phi)$ for some C, ϕ, we simply write $M \Downarrow (C, \phi)$ and whenever $(\mathsf{input}(\emptyset), \emptyset, M) \Uparrow$ we write $M \Uparrow$. In the same guise, we say that M is a *well-typed term with lifted type* α *depending on* $\mathfrak{t}$, and we write $\vdash^{\mathfrak{t}} M : \alpha$, whenever $\emptyset \vdash^{\mathfrak{t}}_{\epsilon} (\mathsf{input}(\emptyset), \emptyset, M) : \alpha; \{\emptyset\}$, while we say that (C, ϕ) is a *well-typed closed configuration with lifted type* α *depending on* $\mathfrak{t}$, and we write $\vdash^{\mathfrak{t}} (C, \phi) : \alpha$, whenever $\emptyset \vdash^{\emptyset}_{\mathfrak{t}} (C, \phi) : \alpha; \mathfrak{t}[\emptyset]_b$. We are now ready to give the relevant type safety results for Proto-Quipper-K.

▶ **Theorem 17** (Subject Reduction). *If* $\vdash^{\mathfrak{t}} M : \alpha$ *and* $\exists C, \phi$ s.t. $M \Downarrow (C, \phi)$, *then* $\vdash^{\mathfrak{t}} (C, \phi) : \alpha$.

Proof sketch. We prove the more general claim that whenever $Q \vdash^{\mathfrak{t}}_{\mathfrak{r}} (D, a, M) : \alpha; \Delta$ and $\exists C, \phi$ s.t. $(D, a, M) \Downarrow (C, \phi)$, then $Q \vdash^{a}_{\lfloor \mathfrak{r}[\mathfrak{t}]^{\{a\}} \rfloor} (C, \phi) : \alpha; \Delta \prec^{a} \mathfrak{t}$, from which we obtain the subject reduction claim by choosing $D = \mathsf{input}(\emptyset), a = \emptyset, Q = \emptyset, \mathfrak{r} = \epsilon$ and $\Delta = \{\emptyset\}$. We proceed by induction on $(D, a, M) \Downarrow (C, \phi)$ and case analysis on the last rule used in its derivation. A number of cases require additional lemmata, more specifically:

1. A substitution lemma is naturally required for the *app, dest, let* cases. In our scenario, this amounts to proving that whenever $\Phi, \Gamma'; Q' \vdash_v V : A$ and Π is a type derivation, then
   a. If the conclusion of Π is $\Phi, \Gamma, x : A; Q \vdash^{\mathfrak{t}}_{c} M : \alpha$, then $\Phi, \Gamma, \Gamma'; Q, Q' \vdash^{\mathfrak{t}}_{c} M[V/x] : \alpha$,
   b. If the conclusion of Π is $\Phi, \Gamma, x : A; Q \vdash_v W : B$, then $\Phi, \Gamma, \Gamma'; Q, Q' \vdash_v W[V/x] : B$,

   We prove the claim separately for the cases in which V has parameter type and linear type. In both cases, we proceed by induction on the size of Π and case analysis on its last rule.
2. The *apply* case is particularly delicate and requires us to prove that all the operations performed by the append function on the applicand and the underlying circuit alter their respective circuit signatures in a predictable way. More specifically:
   a. For the first step of append, we show that equivalent circuits have the same type, that is, that whenever $\emptyset; \emptyset \vdash_v (\vec{\ell}, C, \lambda)_{\mathfrak{t}} : \mathsf{Circ}_{\mathfrak{t}}(T, v)$ and $(\vec{\ell}, C, \lambda)_{\mathfrak{t}} \cong (\vec{\ell'}, C', \lambda')_{\mathfrak{t}}$, then $\emptyset; \emptyset \vdash_v (\vec{\ell'}, C', \lambda')_{\mathfrak{t}} : \mathsf{Circ}_{\mathfrak{t}}(T, v)$. This reflects the idea that the type of a circuit depends on its structure, and not on the specific labels used to convey said structure.
   b. Similarly, for the second step, we show that lifted variable renaming preserves circuit signatures and lifted typing judgments, that is, that for every renaming of lifted variables π, $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$ implies $C\langle\pi\rangle \vdash^{\mathfrak{t}\langle\pi\rangle} Q \triangleright \Delta\langle\pi\rangle$ and $\gamma; \Delta \Vdash^{\mathfrak{t}}_{v} \phi : \alpha$ implies $\gamma\langle\pi\rangle; \Delta\langle\pi\rangle \Vdash^{\mathfrak{t}\langle\pi\rangle}_{v} \phi\langle\pi\rangle : \alpha\langle\pi\rangle$, where γ is a lifted context in $\mathcal{K}_{\mathfrak{t}}(\mathit{CONTEXT})$. Similarly to the previous point, this reflects the idea that the structure of a lifted object is more important than the specific lifted variable names used to convey said structure.
   c. For the third step, we show that whenever we have an underlying circuit C and an applicand D whose labels and lifted variables have already been renamed appropriately, the concatenated circuit $C ::_a D$ is such that its output on branch a contains both the outputs of D and the outputs of C that D was *not* applied on. That is, we prove that whenever $C \vdash^{\mathfrak{t}} Q \triangleright \Delta[Q', Q'']^{\{a\}}$ for some $a \in \mathscr{P}_{\mathfrak{t}}$ and $D \vdash^{\mathfrak{r}} Q' \triangleright \Lambda$ for some D such that the labels that occur in D, but not in Q', are fresh in C and $\mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset$, then $C ::_a D \vdash^{\lfloor \mathfrak{t}[\mathfrak{r}]^{\{a\}} \rfloor} Q \triangleright \lfloor \Delta[\Lambda, Q'']^{\{a\}} \rfloor$. We prove this by induction on $D \vdash^{\mathfrak{r}} Q' \triangleright \Lambda$. This result clearly allows us to conclude the subject reduction claim for the *apply* case.
3. The *let* case is also particularly delicate, although more technical than the *apply* case: it requires us to apply the inductive hypothesis once for the evaluation of the left side of the let and n times for the evaluation of each of the possible branches on the right side, which happens in sequence. This means that the conclusion of each application of the inductive hypothesis must become the premise of the following application. More specifically, for every well-typed right configuration (C_{i+1}, ψ_{a_i}) we must be able to prove that $(C_{i+1}, a \cup a_{i+1}, \mu(a_{i+1})[\phi(a_{i+1})/x])$ is a well-typed left configuration. A key lemma in this process tells us that for any two generic lifted objects $\xi \in \mathcal{K}_{\mathfrak{t}}(X)$ and $\theta \in \mathcal{K}_{\mathfrak{r}}(X)$ and any two assignments $a \in \mathscr{P}_{\mathfrak{t}}$ and $b \in \mathscr{A}_{\mathfrak{r}}$, we have $\lfloor \xi[\theta[x_c]_c^{\mathscr{P}^{b}_{\mathfrak{r}}}]^{\{a\}} \rfloor = \lfloor \xi[\theta]^{\{a\}} \rfloor [x_c]_c^{\mathscr{P}^{a \cup b}_{\lfloor \mathfrak{t}[\mathfrak{r}]^{\{a\}} \rfloor}}$. ◀

▶ **Theorem 18** (Progress). *If* $\vdash^{\mathfrak{t}} M : \alpha$, *then either* $\exists C, \phi$ s.t. $M \Downarrow (C, \phi)$ *or* $M \Uparrow$.

Proof sketch. We consider the equivalent claim that if $\vdash^{\mathfrak{t}} M : \alpha$ and $\nexists C, \phi. M \Downarrow (C, \phi)$, then $M \Uparrow$. We then prove the more general result that if $Q \vdash^{\mathfrak{t}}_{\mathfrak{r}} (D, a, M) : \alpha; \Delta$ and $\nexists C, \phi. (D, a, M) \Downarrow (C, \phi)$, then $(D, a, M) \Uparrow$, from which we obtain the progress claim by choosing $D = \mathsf{input}(\emptyset), a = \emptyset, Q = \emptyset, \mathfrak{r} = \epsilon$ and $\Delta = \{\emptyset\}$. We proceed by coinduction and case analysis on M. The proof is fairly straightforward and makes extensive use of the general subject reduction theorem and of some of its lemmata. The one interesting case is *apply*, which is proven vacuously by showing that append is always defined under the hypothesis:

1. The first step of append must find a circuit $(\vec{k}, D', \lambda')_{\mathfrak{t}} \cong (\vec{\ell}, D, \lambda)_{\mathfrak{t}}$ such that the labels in $\vec{k}$ are given and correspond to the target labels in the underlying circuit C and all the other labels in D' are fresh in C. A key lemma tells us that since $\vec{\ell}$ and $\vec{k}$ have the same M-type T, then it is possible to rename the former to the latter. It is then straightforward to extend this renaming to all the labels occurring in D in a way that fulfills the aforementioned requirements.
2. The second step must compute $D'\langle\pi\rangle$ and $\lambda'\langle\pi\rangle$. This is always possible since π is a valid renaming of lifted variables and thus a permutation.
3. The last step must compute $C ::_a D'\langle\pi\rangle$. We show that for every assignment b occurring in $D'\langle\pi\rangle$ it holds that $\mathsf{dom}(a) \cap \mathsf{dom}(b) = \emptyset$. Therefore, the concatenation operation is defined and append returns $(C ::_a D'\langle\pi\rangle, \lambda'\langle\pi\rangle)$. ◀

Detailed proofs of subject reduction (Theorem 17) and progress (Theorem 18) can be found in the extended version of the paper [2].

6 Conclusion

Our Contributions

This paper introduces a new paradigmatic language for dynamic lifting belonging to the Proto-Quipper family of languages. The language, called Proto-Quipper-K, can be seen as an extension of Proto-Quipper-M which allows for the Boolean information flowing within bit wires to be lifted from the circuit level to the program level. In order to make the circuit construction process as flexible and as general as possible, a powerful type and effect system based on lifting trees has been introduced. This allows for the typing of programs which produce highly non-uniform circuits, making Proto-Quipper-K strictly more expressive than `Quipper` and potentially useful in scenarios such as one-way quantum computing [4]. Although the use of the syntax introduced in this paper is not *essential* to implement the measurement patterns encountered in one-way quantum computing (which, after all, can be simulated by regular quantum circuits), the non-uniform approach to dynamic lifting that we adopt in our work would allow for a higher degree of flexibility in building and manipulating patterns. In other words, even when the circuit produced at the end of the computation is a uniform circuit, we can build it incrementally, going through non-uniform circuits. The main technical results we obtained for Proto-Quipper-K are type soundness, in the sense of subject reduction and progress theorems.

Future Work

In this paper we focused on the operational aspects, leaving an investigation about a possible denotational account of Proto-Quipper-K as future work. A related problem is that of understanding the precise nature of the lifting operation that we use pervasively in the paper. In particular, while it would be tempting to interpret $\mathcal{K}_{\mathfrak{t}}$ as a graded monad [8, 12] with unit $\eta : X \to \mathcal{K}_{\epsilon}(X)$ defined as $\eta\, x = \{x\}$ and multiplication $\lfloor \cdot \rfloor : \mathcal{K}_{\mathfrak{t}}(\mathcal{K}_{\mathfrak{r}}(X)) \to \mathcal{K}_{\mathfrak{t}[\mathfrak{r}]_a}(X)$,

we generally work with objects that do not belong to $\mathcal{K}_{\mathfrak{t}}(\mathcal{K}_{\mathfrak{r}}(X))$ for some fixed $\mathfrak{t}, \mathfrak{r}$, so we find that this interpretation is not entirely appropriate, at least if we assume grades to be elements of *one fixed monoid*.

Another aspect that we left open is the consolidation of homogeneous branches in a computation. In many contexts, it is extremely natural to ask that the type of a term be uniform with respect to a certain lifted variable u, i.e. that it be of the form $u\,\{\alpha\}\{\alpha\}$. If a term M were typed this way, it would be natural to construct a new term $\nu u.M$ in which u is no longer lifted and which could thus be given the type α. We claim that such a local name binder can be added to the language without substantially altering its metatheory.

The last problem that we deliberately leave open concerns the notion of generalized circuits used in this paper. On the one hand, it can certainly be said that it can model quantum circuits in their full generality, including those measurement patterns found in measurement-based quantum computing [4]. On the other hand, while it is clear that such circuits make computational sense (after all, any such computation can be simulated by a classically controlled quantum Turing machine [18]), it is not clear what kind of correspondence exists between them and quantum circuits in their usual form [15], which is the one most often considered in the literature.

Related Work

As already mentioned in the introduction, various paradigmatic λ-calculi modeling the `Quipper` programming language have been introduced in the literature [6, 7, 13, 20, 21]. In this work, we took inspiration from Proto-Quipper-M [20], which however cannot handle dynamic lifting. The only members of the Proto-Quipper family that can handle dynamic lifting, in a uniform and more restricted form than ours, are Proto-Quipper-L [13] and Proto-Quipper-Dyn [6], which have been introduced very recently and independently of this work. Interestingly, the class of circuits targeted by Proto-Quipper-L– which the authors have named *quantum channels* – indeed includes non-uniform circuits like the one in Figure 4, which leads us to believe that they actually match our generalized quantum circuits in terms of expressiveness. However, Proto-Quipper-L's type system rejects the programs that build such circuits. On the other hand, Proto-Quipper-Dyn follows a different approach, with two interleaved operational semantics: one for circuit building, which only happens inside a boxing operator and does *not* support dynamic lifting, and one for circuit execution, where dynamic lifting is allowed. Therefore, as a circuit description language, Proto-Quipper-Dyn targets traditional circuits. At the type level, this distinction is reflected in a modal type system which keeps track of whether dynamic lifting is used, and therefore whether a circuit-building function can be boxed or not. That being said, most of the aforementioned contributions differ from ours in that they focus mainly on the denotational semantics of the language rather than its operational semantics.

The type and effect system paradigm is well known from the literature [3, 12, 14, 16] and has been used in various contexts as a way to reflect information on the effects produced by a program in its type. In our case, the relevant effect is a choice effect, which is mirrored both in the operational semantics and in the type system. As already stated, the problem of giving a proper monadic status to the considered choice effect remains open, although this work shows that *operationally speaking* everything works smoothly.

Finally, it is worth mentioning that the circuit description paradigm is not the only approach to designing quantum programming languages and calculi. For instance, QCL [25], QML [1] and Selinger and Valiron's quantum λ-calculus [23] are some examples of quantum programming languages whose instructions are designed to be executed immediately on quantum hardware, without any direct reference to quantum circuits.

References

1 Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Proc. of LICS*, 2005. doi:10.1109/lics.2005.1.

2 Andrea Colledan and Ugo Dal Lago. On dynamic lifting and effect typing in circuit description languages (extended version), 2022. arXiv:2202.07636.

3 Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Trans. Program. Lang. Syst.*, 41(2), March 2019. doi:10.1145/3293605.

4 Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *J. ACM*, 54(2), April 2007. doi:10.1145/1219092.1219096.

5 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In *Proc. of RC*, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-52482-1_9.

6 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. Proto-quipper with dynamic lifting. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571204.

7 Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages: Extended abstract. In *Proc. of LICS*, 2020. doi:10.1145/3373718.3394765.

8 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In *Proc. of FoSSaCS*, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49630-5_30.

9 Simon J. Gay. Quantum programming languages: Survey and bibliography. *Math. Struct. Comput. Sci.*, 16(4):581–600, August 2006. doi:10.1017/S0960129506005378.

10 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In *Proc. of RC*, pages 110–124, 2013. doi:10.1007/978-3-642-38986-3_10.

11 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper. In *Proc. of PLDI*, pages 333–342, June 2013. doi:10.1145/2499370.2462177.

12 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of POPL*, pages 633–645, January 2014. doi:10.1145/2578855.2535846.

13 Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In *Proc. of FSTTCS*, volume 213 of *LIPIcs*, pages 51:1–51:20, 2021. doi:10.4230/LIPIcs.FSTTCS.2021.51.

14 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited—control-flow algebra and semantics. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, pages 1–32. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-27810-0_1.

15 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.

16 Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design: Recent Insights and Advances*, pages 114–136. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48092-7_6.

17 Jens Palsberg. Toward a universal quantum programming language. *XRDS: Crossroads*, 26(1):14–17, September 2019. doi:10.1145/3355759.

18 Simon Perdrix and Philippe Jorrand. Classically controlled quantum computation. *Math. Struct. Comput. Sci.*, 16(04):601, July 2006. doi:10.1017/s096012950600538x.

19 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

20 Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. In *Proc. of QPL*, volume 266, June 2017. doi:10.4204/EPTCS.266.11.

21 Neil Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.

22 Peter Selinger. A brief survey of quantum programming languages. In *Proc. of FLOPS*, pages 1–6, 2004. doi:10.1007/978-3-540-24754-8_1.

23 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. In *Proc. of TLCA*, pages 354–368, 2005. doi:10.1007/11417170_26.

24 Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016. doi:10.1016/C2014-0-02660-3.

25 Bernhard Ömer. Classical concepts in quantum programming. *Int. J. Theor. Phys.*, 44(7):943–955, July 2005. doi:10.1007/s10773-005-7071-x.

A Type Derivations

$$\text{return}\,\frac{\text{var}\,\dfrac{}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_v a : \mathsf{Qubit}}}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_c^{\epsilon} \mathsf{return}\, a : \{\mathsf{Qubit}\}}$$

Figure 18 Sub-derivation Π', corresponding to the $u = 0$ branch of the lifted computational judgment $\_ : u\,\{\mathbb{1}\}\{\mathbb{1}\}, a : \mathsf{Qubit}; \emptyset \Vdash_c^{u\,\{\{\epsilon\}\}\{\{\epsilon\}\}} u\,\{\mathsf{return}\, a\}\{\mathsf{apply}(\mathit{Meas}_\alpha, a)\} : u\,\{\{\mathsf{Qubit}\}\}\{\{\mathsf{Bit}\}\}$ required by sub-derivation Π of Figure 20.

$$\text{apply}\,\frac{\_ : \mathbb{1}; \emptyset \vdash_v \mathit{Meas}_\alpha : \mathsf{Circ}_\epsilon(\mathsf{Qubit}, \{\mathsf{Bit}\}) \qquad \text{var}\,\dfrac{}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_v a : \mathsf{Qubit}}}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_c^{\epsilon} \mathsf{apply}(\mathit{Meas}_\alpha, a) : \{\mathsf{Bit}\}}$$

Figure 19 Sub-derivation Π'', corresponding to the $u = 1$ branch of the lifted computational judgment $\_ : u\,\{\mathbb{1}\}\{\mathbb{1}\}, a : \mathsf{Qubit}; \emptyset \Vdash_c^{u\,\{\{\epsilon\}\}\{\{\epsilon\}\}} u\,\{\mathsf{return}\, a\}\{\mathsf{apply}(\mathit{Meas}_\alpha, a)\} : u\,\{\{\mathsf{Qubit}\}\}\{\{\mathsf{Bit}\}\}$ required by sub-derivation Π of Figure 20.

$$\text{let}\,\frac{\text{apply}\,\dfrac{\emptyset; \emptyset \vdash_v \mathit{ML} : \mathsf{Circ}_{u\,\{\epsilon\}\{\epsilon\}}(\mathsf{Qubit}, u\,\{\mathbb{1}\}\{\mathbb{1}\}) \qquad \text{var}\,\dfrac{}{q : \mathsf{Qubit}; \emptyset \vdash_v q : \mathsf{Qubit}}}{q : \mathsf{Qubit}; \emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{apply}_u(\mathit{ML}, q) : u\,\{\mathbb{1}\}\{\mathbb{1}\}} \quad \Pi' \quad \Pi''}{\begin{array}{l} u\,\{\mathsf{return}\, a\}\{\mathsf{apply}(\mathit{Meas}_\alpha, a)\} \in \mathcal{K}_{u\,\{\epsilon\}\{\epsilon\}}(\mathit{TERM}) \\ q : \mathsf{Qubit}, a : \mathsf{Qubit}; \emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{let}\ \_ = \mathsf{apply}_u(\mathit{ML}, q)\ \mathsf{in} \\ \qquad\qquad u\,\{\mathsf{return}\, a\}\{\mathsf{apply}(\mathit{Meas}_\alpha, a)\} : u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\} \end{array}}$$

Figure 20 Sub-derivation Π, where the conclusion is the expansion of the (trivial) lifted computational judgment required by the *let* rule in Figure 21.

$$
\text{abs}\;\dfrac{
\text{return}\;\dfrac{
\text{abs}\;\dfrac{
\text{let}\;\dfrac{
\text{apply}\;\dfrac{\emptyset;\emptyset \vdash_v H : \mathsf{Circ}_\epsilon(\mathsf{Qubit},\{\mathsf{Qubit}\}) \qquad \text{var}\;\dfrac{}{q : \mathsf{Qubit};\emptyset \vdash_v q : \mathsf{Qubit}}}{q : \mathsf{Qubit};\emptyset \vdash_c^{\epsilon} \mathsf{apply}(H,q) : \{\mathsf{Qubit}\}} \qquad \Pi
\qquad \{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} \in \mathcal{K}_\epsilon(TERM)
}{
a : \mathsf{Qubit}, q : \mathsf{Qubit};\emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}
}
}{
q : \mathsf{Qubit};\emptyset \vdash_v \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : \mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}
}
}{
q : \mathsf{Qubit};\emptyset \vdash_c^{\epsilon} \mathsf{return}\ \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : \{\mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}\}
}
}{
\emptyset;\emptyset \vdash_v \lambda q_{\mathsf{Qubit}}.\,\mathsf{return}\ \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : \mathsf{Qubit} \multimap \{\mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}\}
}
$$

Figure 21 Type derivation for the Proto-Quipper-K program in Figure 14. For brevity, H, $Meas_\alpha$ and ML stand for the corresponding boxed circuits employed in Figure 14 and arrow annotations are omitted. Sub-derivation Π is given in Figure 20.