Circuit Width Estimation via Effect Typing and Linear Dependency

Anonymous Authors

Abstract. Circuit description languages are a class of quantum programming languages in which programs are classical and produce a *description* of a quantum computation, in the form of a *quantum circuit*. Since these programs can leverage all the expressive power of high-level classical languages, circuit description languages have been successfully used to describe complex and practical quantum algorithms, whose circuits, however, may involve many more qubits and gate applications than current quantum architectures can actually muster. In this paper, we present Proto-Quipper-R, a circuit description language endowed with a linear dependent type-and-effect system capable of deriving parametric upper bounds on the width of the circuits produced by a program. We prove both the standard type safety results and that the resulting resource analysis is correct with respect to a big-step operational semantics. We also show that our approach is expressive enough to verify realistic quantum algorithms.

Keywords: Effect Typing · Lambda Calculus · Quantum Computing · Quipper

1 Introduction

With the promise of providing efficient algorithmic solutions to many problems [28, 32, 12], some of which are traditionally believed to be intractable [54], quantum computing is the subject of intense investigation by various research communities within computer science, not least that of programming language theory [43, 25, 51]. Various proposals for idioms capable of tapping into this new computing paradigm have appeared in the literature since the late 1990s. Some of these approaches turn out to be fundamentally new [52, 1, 49], while many others are strongly inspired by classical languages and traditional programming paradigms [61, 53, 48, 44].

One of the major obstacles to the practical adoption of quantum algorithmic solutions is the fact that despite huge efforts by scientists and engineers alike, it seems that reliable quantum hardware, contrary to classical one, does not scale too easily: although quantum architectures with up to a couple hundred qubits have recently seen the light [10, 39, 11], it is not yet clear whether the so-called quantum advantage [45] is a concrete possibility, given the tremendous challenges posed by the quantum decoherence problem [50].

This entails that software which makes use of quantum hardware must be designed with great care: whenever part of a computation has to be run on quantum hardware, the amount of resources it needs, and in particular the amount of qubits it uses, should be kept to a minimum. More generally, a fine control over the low-level aspects of the computation, something that we willingly abstract from in most cases when dealing with classical computations, should be exposed to the programmer in the quantum case. This, in turn, has led to the development and adoption of many domain-specific programming languages and libraries in which the programmer *explicitly* manipulates qubits and quantum circuits, while still making use of all the features of a high-level classical programming language. This is the case of the `Qiskit` and `Cirq` libraries [18], but also of the `Quipper` language [26, 27].

At the fundamental level, `Quipper` is a circuit description language embedded in `Haskell`. Because of this, `Quipper` inherits all the expressiveness of the high level, higher-order functional programming language that is its host, but for the same reason it also lacks a formal semantics. Nonetheless, over the past few years, a number of calculi, collectively known as the Proto-Quipper language family, have been developed to formalize interesting fragments and extensions of `Quipper` and its type system [48, 46]. Extensions include, among others, dynamic lifting [36, 22, 9] and dependent types [23, 21], but resource analysis is still a rather unexplored research direction in the Proto-Quipper community [56].

The goal of this work is to show that type systems indeed enable the possibility of reasoning about the size of the circuits produced by a Proto-Quipper program. Specifically, we show how linear dependent types in the form given by Gaboardi and Dal Lago [13, 24, 15, 16] can be adapted to Proto-Quipper, allowing to derive upper bounds on circuit widths that are parametric on the circuit's input size, that is to say, the number of input wires to the circuit, be they classical or quantum. This enables a form of static analysis of the resource consumption of circuit families and, consequently, of the quantum algorithms described in the language. Technically, a key ingredient of this analysis, besides linear dependency, is a novel form of effect typing in which the quantitative information coming from linear dependency informs the effect system and allows it to keep circuit widths under control.

The rest of the paper is organized as follows. Section 2 informally explores the problem of estimating the width of circuits produced by `Quipper`, while also introducing the language. Section 3 provides a more formal definition of the Proto-Quipper language. In particular, it gives an overview of the system of simple types due to Selinger and Rios [46], which however is not meant to reason about the size of circuits. We then move on to the most important technical contribution of this work, namely the linear dependent and effectful type system, which is introduced in Section 4 and proven to guarantee both type safety and a form of total correctness in Section 5. Section 6 is dedicated to an example of a practical application of our type and effect system, that is, a program that builds the Quantum Fourier Transform (QFT) circuit [12, 40] and which is verified to do so without any ancillary qubits.

2 An Overview on Circuit Width Estimation

Quipper allows programmers to describe quantum circuits in a high-level and elegant way, using both gate-by-gate and circuit transformation approaches. Quipper also supports hierarchical and parametric circuits, thus promoting a view in which circuits become first-class citizens. Quipper has been shown to be scalable, in the sense that it has been effectively used to describe complex quantum algorithms that easily translate to circuits involving trillions of gates applied to millions of qubits. The language allows the programmer to optimize the circuit, e.g. by using ancilla qubits for the sake of reducing the circuit depth, or recycling qubits that are no longer needed.

One feature that Quipper lacks is a methodology for *statically* proving that important parameters — such as the the width — of the underlying circuit are below certain limits, which of course would need to be parametric on the input size of the circuit. If this kind of analysis were available, then it would be possible to derive bounds on the number of qubits needed to solve any instance of a problem, and ultimately to know in advance how big of an instance can be *possibly* solved given a fixed amount of qubits.

In order to illustrate the kind of scenario we are reasoning about, this section offers some simple examples of Quipper programs, showing in what sense we can think of capturing the quantitative information that we are interested in through types and effect systems and linear dependency. We proceed at a very high level for now, without any ambition of formality.

Let us start with the example of Figure 1. The Quipper function on the left builds the quantum circuit on the right: an (admittedly contrived) implementation of the quantum not operation. The dumbNot function implements negation using a controlled not gate and an ancillary qubit a, which is initialized and discarded within the body of the function. This qubit does not appear in the interface of the circuit, but it clearly adds to its overall width, which is 2.

```
dumbNot :: Qubit -> Circ Qubit
dumbNot q = do
 a <- qinit True
 (q,a) <- controlled_not q a
 qdiscard a
 return q
```

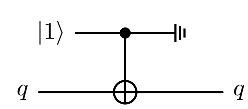

Fig. 1. An implementation of the quantum not operation using an ancilla.

Consider now the higher order function in Figure 2. This function takes as input a circuit building function f, an integer n and describes the circuit obtained by applying f's circuit n times to the input qubit q. It is easy to see that the width of the circuit produced in output by iter dumbNot n is equal to 2, even though, overall, the number of qubits initialized during the computation is equal to n. The point is that each ancilla is created only *after* the previous one has been discarded, thus enabling a form of qubit recycling.

```
iter :: (Qubit -> Circ Qubit)
-> Int -> Qubit -> Circ Qubit
iter f 0 q = return q
iter f n q = do
 q <- f q
 iter f (n-1) q
```

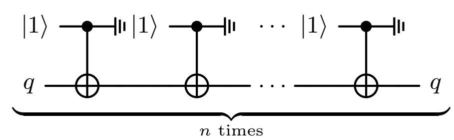

Fig. 2. A higher-order function which iterates a circuit-building function `f` on an input qubit `q` and the result of its application to the `dumbNot` function from Figure 1.

Is it possible to statically analyze the width of the circuit produced in output by `iter dumbNot n` so as to conclude that it is constant and equal to 2? What techniques can we use? Certainly, the presence of higher order types complicates the problem, already in itself non-trivial. The approach we propose in this paper is based on two ingredients. The first is the so-called effect typing [41]. In this context the effect produced by the program is nothing more than the circuit and therefore it is natural to think of an effect system in which the width of such circuit, and only that, is exposed. Therefore, the arrow type $A \to B$ should be decorated with an expression indicating the width of the circuit produced by the corresponding function when applied to an argument of type A. Of course, the width of an individual circuit is a natural number, so it would make sense to annotate the arrow with a natural. For technical reasons, however, it will also be necessary to keep track of another natural number, corresponding to the number of wire resources that the function captures from the surrounding environment. This necessity stems from a need to keep close track of wires even in the presence of data hiding, and will be explained in further detail in Section 4.

Under these premises, the `dumbNot` function would receive type $\mathsf{Qubit} \to_{2,0} \mathsf{Qubit}$, meaning that it takes as input a qubit and produces a circuit of width 2 which outputs a qubit. Note that the second annotation is 0, since we do not capture anything from the function's environment, let alone a wire. Consequently, because `iter` iterates in sequence and because the ancillary qubit in `dumbNot` can be reused, the type of `iter dumbNot n` would also be $\mathsf{Qubit} \to_{2,0} \mathsf{Qubit}$.

Let us now consider a slightly different situation, in which the width of the produced circuit is not constant, but rather increases proportionally to the circuit's input size. Figure 3 shows a `Quipper` function that returns a circuit on n qubits in which the Hadamard gate is applied to each qubit. This simple circuit represents the preprocessing phase of the Deutsch-Josza algorithm [8]. It is obvious that this function works on inputs of arbitrary size, and therefore we can interpret it as a circuit family, parametrized on the length of the input list of qubits. This quantity, although certainly natural, is unknown statically and corresponds precisely to the width of the produced circuit. A question therefore arises as to whether the kind of effect typing we briefly hinted at in the previous

```
hadamardN :: [Qubit] -> Circ [Qubit]
hadamardN [] = return []
hadamardN (q:qs) = do
 q <- hadamard q
 qs <- hadamardN qs
 return (q:qs)
```

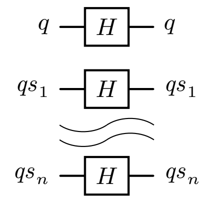

Fig. 3. The `hadamardN` function implements a circuit family where circuits have width linear in their input size.

paragraph is capable of dealing with such a function. Certainly, the expressions used to annotate arrows cannot be, like in the previous case, mere *constants*, as they clearly depend on the size of the input list. Is there a way to reflect this dependency in types? Certainly, one could go towards a fully-fledged notion of dependent types, like the ones proposed in [23], but a simpler approach, in the style of Dal Lago and Gaboardi's linear dependent types [13, 24, 15, 16] turns out to be enough for this purpose. This is precisely the route that we follow in this paper. In this approach, terms can indeed appear in types, but that is only true for a very restricted class of terms, disjoint from the ordinary ones, called *index terms*. As an example, the type of the function `hadamardN` above could become $\mathsf{List}^i\,\mathsf{Qubit} \rightarrow_{i,0} \mathsf{List}^i\,\mathsf{Qubit}$, where i is an *index variable*. The meaning of the type would thus be that `hadamardN` takes as input any list of qubits of length i and produces a circuit of width at most i which outputs i qubits. The language of indices is better explained in Section 4, but in general we can say that indices are arithmetical expressions over natural numbers and index variables, and can thus express non-trivial dependencies between input sizes and corresponding circuit widths.

3 The Proto-Quipper Language

This section aims at introducing the Proto-Quipper family of calculi to the non-specialist, without any form of resource analysis. At its core, Proto-Quipper is a linear lambda calculus with bespoke constructs to build and manipulate circuits. Circuits are built as a side-effect of a computation, behind the scenes, but they can also appear and be manipulated as data in the language.

Types	$TYPE$	$A, B ::= \mathbb{1} \mid w \mid\, !A \mid A \otimes B \mid A \multimap B \mid \mathsf{List}\,A \mid \mathsf{Circ}(T, U)$
Parameter types	$PTYPE$	$P, R ::= \mathbb{1} \mid\, !A \mid P \otimes R \mid \mathsf{List}\,P \mid \mathsf{Circ}(T, U)$
Bundle types	$BTYPE$	$T, U ::= \mathbb{1} \mid w \mid T \otimes U$

Fig. 4. The Proto-Quipper types.

The types of Proto-Quipper are given in Figure 4. Speaking at a high level, we can say that in Proto-Quipper types are generally linear. In particular, $w \in \{\mathsf{Bit}, \mathsf{Qubit}\}$ is a *wire type* and is linear, while $\multimap$ is the linear arrow constructor. A subset of types, called *parameter types*, represent the values of the language that are not linear and that can therefore be copied. Any term of type A can be *lifted* into a duplicable parameter of type $!\,A$ if its type derivation does not require the use of linear resources.

Terms	*TERM*	$M, N ::= V\,W \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\,V \mid \mathsf{box}_T\,V$ $\mid \mathsf{apply}(V, W) \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$
Values	*VAL*	$V, W ::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\,M \mid (\bar{\ell}, \mathcal{C}, \bar{k}) \mid \langle V, W\rangle$ $\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}\ V\ W$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k} ::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle$

Fig. 5. The Proto-Quipper syntax.

Now, let us informally dissect the language as presented in Figure 5, starting with the language of values. The main constructs of interest are *labels* and *boxed circuits*. A label ℓ represents a reference to a free wire of the underlying circuit being built and is attributed a wire type $w \in \{\mathsf{Bit}, \mathsf{Qubit}\}$. Due to the no-cloning property of quantum states [40], labels have to be treated linearly. Arbitrary structures of labels form a subset of values which we call *wire bundles* and which are given *bundle types*. On the other hand, a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ represents a circuit object $\mathcal{C}$ as a datum within the language, together with its input and output interfaces $\bar{\ell}$ and $\bar{k}$. Such a value is given type $\mathsf{Circ}(T, U)$, where bundle types T and U are the input and output types of the circuit, respectively. Boxed circuits can be copied, manipulated by primitive functions and, more importantly, applied to the underlying circuit. This last operation, which lies at the core of Proto-Quipper's circuit-building capabilities, is possible thanks to the apply operator. This operator takes as first argument a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ and appends $\mathcal{C}$ to the underlying circuit $\mathcal{D}$. How does apply know *where* exactly in $\mathcal{D}$ to apply $\mathcal{C}$? Thanks to a second argument: a bundle of wires $\bar{t}$ coming from the free output wires of $\mathcal{D}$, which identify the exact location where $\mathcal{C}$ is supposed to be attached.

The language is expected to be endowed with constant boxed circuits corresponding to fundamental gates (e.g. Hadamard, controlled not, etc.), but the programmer can also introduce their own boxed circuits via the box operator. Intuitively, box takes as input a circuit-building function and executes it in a sandboxed environment, on dummy arguments, in a way that leaves the underlying circuit unchanged. Said function produces a standalone circuit $\mathcal{C}$, which is then returned by the box operator as a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$.

Figure 6 shows the Proto-Quipper term corresponding to the `Quipper` program in Figure 1, as an example of the use of the language. Note that $\mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N$ is syntactic sugar for $\mathsf{let}\ z = M\ \mathsf{in}\ \mathsf{let}\ \langle x, y\rangle = z\ \mathsf{in}\ N$. The *dumbNot* function is given type $\mathsf{Qubit} \multimap \mathsf{Qubit}$ and builds the circuit shown in Figure 1 when applied to an argument.

$$
\begin{aligned}
\mathit{dumbNot} \triangleq \lambda q_{\mathsf{Qubit}}.\ & \mathsf{let}\ a = \mathsf{apply}(\mathsf{INIT}_1, *)\ \mathsf{in} \\
& \mathsf{let}\ \langle q, a\rangle = \mathsf{apply}(\mathsf{CNOT}, \langle q, a\rangle)\ \mathsf{in} \\
& \mathsf{let}\ \_ = \mathsf{apply}(\mathsf{DISCARD}, a)\ \mathsf{in} \\
& \mathsf{return}\ q
\end{aligned}
$$

Fig. 6. An example Proto-Quipper program. $\mathsf{INIT}_1, \mathsf{CNOT}$ and $\mathsf{DISCARD}$ are primitive boxed circuits implementing the corresponding elementary operations.

On the classical side of things, it is worth mentioning that Proto-Quipper as presented in this section does *not* support general recursion. A limited form of recursion on lists is instead provided via a primitive fold constructor, which takes as argument a (copiable) step function of type $!((B \otimes A) \multimap B)$, an initial value of type B, and constructs a function of type $\mathsf{List}\ A \multimap B$. Although this workaround is not enough to recover the full power of general recursion, it appears that it is enough to describe many quantum algorithms. Figure 7 shows an example of the use of fold to reverse a list. Note that $\lambda\langle x, y\rangle_{A\otimes B}.M$ is syntactic sugar for $\lambda z_{A\otimes B}.\mathsf{let}\ \langle x, y\rangle = z\ \mathsf{in}\ M$.

$$
\mathit{rev} \triangleq \mathsf{fold}\ \mathsf{lift}(\lambda\langle \mathit{revList}, q\rangle_{\mathsf{List\ Qubit}\otimes\mathsf{Qubit}}.\mathsf{return}\ (\mathsf{cons}\ q\ \mathit{revList}))\ \mathsf{nil}
$$

Fig. 7. An example of the use of fold: the function that reverses a list.

To conclude this section, we just remark how all of the `Quipper` programs shown in Section 2 can be encoded in Proto-Quipper. However, Proto-Quipper's system of simple types in unable to tell us anything about the resource consumption of these programs. Of course, one could run `hadamardN` on a concrete input and examine the size of the circuit produced at run-time, but this amounts to *testing*, not *verifying* the program, and lacks the qualities of staticity and parametricity that we seek.

4 Incepting Linear Dependency and Effect Typing

We are now ready to expand on the informal definition of the Proto-Quipper language given in Section 3, to reach a formal definition of Proto-Quipper-R: a linearly and dependently typed language whose type system supports the derivation of upper bounds on the width of the circuits produced by programs.

4.1 Types and Syntax of Proto-Quipper-R

$$
\begin{array}{llrcl}
\text{Types} & \mathit{TYPE} & A, B & ::= & \mathbb{1} \mid w \mid\, !\,A \mid A \otimes B \mid A \multimap_{I,J} B \mid \mathsf{List}^I\, A \mid \mathsf{Circ}^I(T, U) \\
\text{Param. types} & \mathit{PTYPE} & P, R & ::= & \mathbb{1} \mid\, !\,A \mid P \otimes R \mid \mathsf{List}^I\, P \mid \mathsf{Circ}^I(T, U) \\
\text{Bundle types} & \mathit{BTYPE} & T, U & ::= & \mathbb{1} \mid w \mid T \otimes U \mid \mathsf{List}^I\, T, \\
 & & & & \\
\text{Terms} & \mathit{TERM} & M, N & ::= & V\,W \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\, V \mid \mathsf{box}_T\, V \\
 & & & & \mid \mathsf{apply}(V, W) \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \\
\text{Values} & \mathit{VAL} & V, W & ::= & * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\bar{\ell}, \mathcal{C}, \bar{k}) \mid \langle V, W\rangle \\
 & & & & \mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}_i\ V\ W \\
\text{Wire bundles} & \mathit{BVAL} & \bar{\ell}, \bar{k} & ::= & * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{cons}\ \bar{\ell}\ \bar{k} \\
\text{Indices} & \mathit{INDEX} & I, J & ::= & i \mid n \mid I + J \mid I - J \mid I \times J \mid \mathsf{max}(I, J) \mid \mathsf{max}_{i<I}\, J
\end{array}
$$

Fig. 8. Proto-Quipper-R syntax and types.

The types and syntax of Proto-Quipper-R are given in Figure 8. As we mentioned, one of the key ingredients of our type system are the index terms with which we annotate standard Proto-Quipper types. These indices provide quantitative information about the elements of the resulting types, in a manner reminiscent of refinement types [19, 47]. In our case, we are primarily concerned with circuit width, which means that the natural starting point of our extension of Proto-Quipper is precisely the circuit type: $\mathsf{Circ}^I(T, U)$ has elements the boxed circuits of input type T, output type U, *and width bounded by* I. Term I is precisely what we call an index, that is, an arithmetical expression denoting a natural value. Looking at the grammar for indices, their interpretation is fairly straightforward, with a few notes: n is a natural number, i is an index variable, $I - J$ denotes *natural* subtraction, such that $I - J = 0$ whenever $I \leq J$, and lastly $\mathsf{max}_{i<I}\, J$ is the maximum for i going from 0 (included) to I (excluded) of J, where i can occur free in J. Note that $I = 0$ implies $\mathsf{max}_{i<I}\, J = 0$. While the index in a circuit type denotes an upper bound, the index in a type of the form $\mathsf{List}^I\, A$ denotes the *exact* length of the lists of that type. While this refinement is quite restrictive in a generic scenario, it allows us to include lists of labels among wire bundles, since they are effectively isomorphic to finite tensors of labels and therefore represent wire bundles of known size. Lastly, as we anticipated in Section 2, an arrow type $A \multimap_{I,J} B$ is annotated with *two* indices: I is an upper bound to the width of the circuit built by the function once it is applied to an argument of type A, while J describes the exact number of wire resources captured in the function's closure. The utility of this last annotation will be clearer in Section 4.3.

The languages for terms and values are almost the same as in Proto-Quipper, with the minor difference that the fold operator now binds the index variable name i within its first argument. This variable appears locally in the type of the step function, in such a way as to allow each iteration of the fold to contribute to the overall circuit width in a *different* way.

4.2 A Formal Language for Circuits

The type system of Proto-Quipper-R is designed to reason about the width of circuits. Therefore, before we formally introduce the type system in Section 4.3, we ought to introduce circuits themselves in a formal way. So far, we have only spoken of circuits at a very high and intuitive level, and we have represented them only graphically. Looking at the circuits in Section 2, what do they have in common? At the fundamental level, they are made up of elementary operations applied to specific wires. Of course, the order of these operations matters, as does the order of wires that they are applied to.

In the existing literature on Proto-Quipper, circuits are usually interpreted as morphisms in a symmetric monoidal category [46], but this approach makes it particularly hard to reason about their intensional properties, such as precisely width. For this reason, we opt for a *concrete* model of wires and circuits, rather than an abstract one.

Luckily, we already have a datatype modeling ordered structures of wires, that is, the wire bundles that we introduced in the previous sections. We use them as the basis upon which we build circuits.

Wire bundles	$BVAL$	$\bar{\ell}, \bar{k} ::= * \mid \ell \mid \langle \bar{\ell}, \bar{k} \rangle \mid \mathsf{nil} \mid \mathsf{cons}\ \bar{\ell}\ \bar{k},$
Bundle types	$BTYPE$	$T, U ::= \mathbb{1} \mid w \mid T \otimes U \mid \mathsf{List}^I\, T,$
Circuits	$CIRC$	$\mathcal{C}, \mathcal{D} ::= id_Q \mid \mathcal{C}; g(\bar{\ell}) \to \bar{k}.$

Fig. 9. CRL syntax and types.

That being said, Figure 9 introduces the Circuit Representation Language (CRL) which we use as the target for circuit building in Proto-Quipper-R. Wire bundles are exactly as in Figure 8 and represent arbitrary structures of wires, while circuits themselves are defined very simply as a sequence of elementary operations applied to said structures. We call Q a *label context* and define it as a mapping from label names to wire types. We use label contexts as a mean to keep track of the set of labels available in a computation, alongside their respective types. Circuit id_Q represents the identity circuit taking as input the labels in Q and returning them unchanged, while $\mathcal{C}; g(\bar{\ell}) \to \bar{k}$ represents the application of the elementary operation g to the wires identified by $\bar{\ell}$ among the outputs of $\mathcal{C}$. Operation g outputs the wire bundle $\bar{k}$, whose labels become part of the outputs of the circuit as a whole. Note that an "elementary operation" is usually the application of a gate, but it could also be a measurement, or the initialization or discarding of a wire. Although semantically very different, from the perspective of circuit building these operations are just elementary building blocks in the construction of a more complex structure, and it makes no sense to distinguish between them syntactically. Circuits are amenable to a form of concatenation. We write the *concatenation* of $\mathcal{C}$ and $\mathcal{D}$ as $\mathcal{C} :: \mathcal{D}$ and define it in the natural way, that is, $\mathcal{C}$ followed by all the elementary operations occurring in $\mathcal{D}$. Note that $\mathcal{C} :: id_Q = \mathcal{C} = id_Q :: \mathcal{C}$, for all $\mathcal{C}, Q$.

Circuit Typing Naturally, not all circuits built from the CRL syntax make sense. For example $id_{(\ell:\mathsf{Qubit})}; H(k) \to k$ and $id_{(\ell:\mathsf{Qubit})}; CNOT(\langle \ell, \ell \rangle) \to \langle k, t \rangle$ are both syntactically correct, but the first applies a gate to a non-existing wire, while the second violates the no-cloning theorem by duplicating ℓ. To rule out such ill-formed circuits, we employ a rudimentary type system for circuits which allows us to derive judgments of the form $\mathcal{C} : Q \to L$, which informally read "circuit $\mathcal{C}$ is well-typed with input label context Q and output label context L".

$$\mathit{unit}\,\frac{}{\emptyset \vdash_w * : \mathbb{1}} \qquad \mathit{lab}\,\frac{}{\ell : w \vdash_w \ell : w} \qquad \mathit{nil}\,\frac{\vDash I = 0}{\emptyset \vdash_w \mathsf{nil} : \mathsf{List}^I\, T}$$

$$\mathit{pair}\,\frac{Q_1 \vdash_w \bar{\ell} : T \qquad Q_2 \vdash_w \bar{k} : U}{Q_1, Q_2 \vdash_w \langle \bar{\ell}, \bar{k} \rangle : T \otimes U}$$

$$\mathit{cons}\,\frac{Q_1 \vdash_w \bar{\ell} : T \qquad Q_2 \vdash_w \bar{k} : \mathsf{List}^J\, T \qquad \vDash I = J + 1}{Q_1, Q_2 \vdash_w \mathsf{cons}\, \bar{\ell}\, \bar{k} : \mathsf{List}^I\, T}$$

$$\mathit{id}\,\frac{}{id_Q : Q \to Q} \qquad \mathit{seq}\,\frac{\mathcal{C} : Q \to L, H \quad H \vdash_w \bar{\ell} : T \quad K \vdash_w \bar{k} : U \quad g \in \mathscr{G}(T, U)}{\mathcal{C}; g(\bar{\ell}) \to \bar{k} : Q \to L, K}$$

Fig. 10. The CRL type system.

The typing rules for CRL are given in Figure 10. We call $Q \vdash_w \bar{\ell} : T$ a *wire judgment*, and we use it to give a structured type to an otherwise unordered label context, by means of a wire bundle. Most rules are straightforward, except those for lists, which rely on a judgment of the form $\vDash I = J$. This is to be intended as a semantic judgment asserting that I and J are closed and equal when interpreted as natural numbers. Within the rule, this reflects the idea that there are many ways to syntactically represent the length of a list. For example, nil can be given type $\mathsf{List}^0\, T$, but also $\mathsf{List}^{1-1}\, T$ or $\mathsf{List}^{0\times 5}\, T$. This kind of flexibility might seem unwarranted for such a simple language, but it is useful to effectively interface CRL and the more complex Proto-Quipper-R. Speaking of the actual circuit judgments, the *seq* rule tells us that the the application of an elementary operation g is well typed whenever g only acts on labels occurring in the outputs of $\mathcal{C}$ (those in $\bar{\ell}$, that is in H), produces in output labels that do not clash with the remaining outputs of $\mathcal{C}$ (since L, K denotes the disjoint union of the two label contexts) and is of the right type. This last requirement is expressed as $g \in \mathscr{G}(T, U)$, where $\mathscr{G}(T, U)$ is the subset of elementary operations that can be applied to an input of type T to obtain an output of type U. For example, the Hadamard gate, which acts on a single qubit, is in $\mathscr{G}(\mathsf{Qubit}, \mathsf{Qubit})$.

Circuit Width Among the many properties of circuits, we are interested in width, so we conclude this section by giving a formal status to this quantity. As we saw in Section 2, when we initialize a new wire, we can reuse previously discarded wires in such a way that the width of a circuit is not always equal to the number of wires that are initialized. We formalize this intuition in the following definition.

Definition 1 (Circuit Width). *We define the width of a CRL circuit $\mathcal{C}$, written* $\mathrm{width}(\mathcal{C})$*, as follows*

$$\mathrm{width}(id_Q) = |Q| \tag{1}$$

$$\mathrm{width}(\mathcal{C}; g(\bar{\ell}) \to \bar{k}) = \mathrm{width}(\mathcal{C}) + \max(0, \mathrm{new}(g) - \mathrm{discarded}(\mathcal{C})) \tag{2}$$

where $|Q|$ is the number of labels in Q, $\mathrm{new}(g)$ represents the net number of new wires initialized by g, and $\mathrm{discarded}(\mathcal{C})$ is the number of wires that have been effectively discarded by the end of $\mathcal{C}$, obtained as the difference between $\mathcal{C}$'s width and the number of its outputs. The idea is that whenever we require a new wire in our computation, first we try to reuse a previously discarded wire, in which case the initialization does not add to the total width of the circuit ($\mathrm{new}(g) \leq \mathrm{discarded}(\mathcal{C})$), and *only if we cannot do so* we actually create a new wire, increasing the overall width of the circuit ($\mathrm{new}(g) > \mathrm{discarded}(\mathcal{C})$).

Now that we have a formal definition of circuit types and width, we can state a fundamental property of the concatenation of well-typed circuits, which is illustrated in Figure 11 and proven in Theorem 1. We use this result pervasively in proving the correctness of Proto-Quipper-R in section 5.

Theorem 1 (CRL). *Given $\mathcal{C} : Q \to L, H$ and $\mathcal{D} : H \to K$ such that the labels shared by $\mathcal{C}$ and $\mathcal{D}$ are all and only those in H, we have*

1. $\mathcal{C} :: \mathcal{D} : Q \to L, K$,
2. $\mathrm{width}(\mathcal{C} :: \mathcal{D}) \leq \max(\mathrm{width}(\mathcal{C}), \mathrm{width}(\mathcal{D}) + |L|)$.

Proof. By induction of the derivation of $\mathcal{D} : H \to K$.

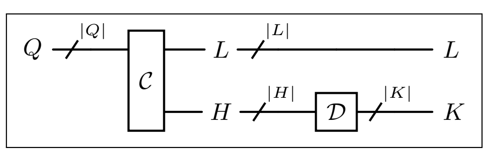

Fig. 11. The kind of scenario described by Theorem 1.

4.3 Typing Programs

Going back to Proto-Quipper-R, we have already seen how the standard Proto-Quipper types are refined with quantitative information. However, decorating types is not enough for the purposes of width estimation. Recall that, in general, a Proto-Quipper program produces a circuit as a *side effect* of its evaluation. If we want to reason about the width of said circuit, it is not enough to rely on a regular linear type system, although dependent. Rather, we have to introduce the second ingredient of our analysis and turn to a *type-and-effect system* [41], revolving around a type judgment of the form

$$\Theta; \Gamma; Q \vdash_c M : A; I, \tag{3}$$

which intuitively reads "for all natural values of the index variables in Θ, under typing context Γ and label context Q, term M has type A and produces a circuit of width at most I". Therefore, Θ is a collection of index variables which are universally quantified in the rest of the judgment, while Γ is a typing context for parameter and linear variables alike. When a typing context contains exclusively parameter variables, we write it as Φ. In this judgment, I plays the role of an *effect annotation*, describing a relevant aspect of the side effect produced by the evaluation of M (i.e. the width of the produced circuit). The attentive reader might wonder why this annotation consists only of one index, whereas when we discussed arrow types in previous sections we needed two. The reason is that the second index, which we use to keep track of the number of wires captured by a function, is redundant in a typing judgment where the same quantity can be inferred directly from the environments Γ and Q. A similar typing judgment is introduced for values, which are effect-less:

$$\Theta; \Gamma; Q \vdash_v V : A. \tag{4}$$

The rules for deriving typing judgments are those in Figure 12, where Γ_1, Γ_2 denotes the union of two contexts with disjoint domains. A well-formedness judgment of the form $\Theta \vdash I$ means that all the free index variables occurring in I are in Θ. Well-formedness is lifted to types and typing contexts in the natural way. Among interesting typing rules, we can see how the *circ* rule bridges between CRL and Proto-Quipper-R. A boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ is well typed with type $\mathsf{Circ}^I(T, U)$ when $\mathcal{C}$ is no wider than the quantity denoted by I, $\mathcal{C} : Q \to L$ and $\bar{\ell}, \bar{k}$ contain all and only the labels in Q and L, respectively, acting as a language-level interface to $\mathcal{C}$.

The two main constructs that interact with circuits are apply and box. The *apply* rule is the foremost place where effects enter the type derivation: V represents some boxed circuit of width at most I, so its application to an appropriate wire bundle W produces exactly a circuit of width at most I. The *box* rule, on the other hand, works approximately in the opposite direction. If V is a circuit building function that, once applied to an input of type T, would build a circuit of output type U and width at most I, then boxing it means turning it into a boxed circuit with the same characteristics. Note that the *box* rule requires that

$$unit\ \frac{\Theta \vdash \Phi}{\Theta;\Phi;\emptyset \vdash_v * : \mathbb{1}} \qquad lab\ \frac{\Theta \vdash \Phi}{\Theta;\Phi;\ell : w \vdash_v \ell : w}$$

$$var\ \frac{\Theta \vdash \Phi, x : A}{\Theta;\Phi, x : A;\emptyset \vdash_v x : A} \qquad abs\ \frac{\Theta;\Gamma, x : A;Q \vdash_c M : B;I}{\Theta;\Gamma;Q \vdash_v \lambda x_A.M : A \multimap_{I,\#(\Gamma;Q)} B}$$

$$app\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : A \multimap_{I,J} B \qquad \Theta;\Phi,\Gamma_2;Q_2 \vdash_v W : A}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c V\,W : B;I}$$

$$lift\ \frac{\Theta;\Phi;\emptyset \vdash_c M : A;0}{\Theta;\Phi;\emptyset \vdash_v \mathsf{lift}\,M : !\,A} \qquad force\ \frac{\Theta;\Phi;\emptyset \vdash_v V : !\,A}{\Theta;\Phi;\emptyset \vdash_c \mathsf{force}\,V : A;0}$$

$$circ\ \frac{\mathcal{C} : Q \to L \qquad Q \vdash_w \bar{\ell} : T \qquad L \vdash_w \bar{k} : U \qquad \Theta \vDash \mathrm{width}(\mathcal{C}) \leq I \qquad \Theta \vdash \Phi}{\Theta;\Phi;\emptyset \vdash_v (\bar{\ell},\mathcal{C},\bar{k}) : \mathsf{Circ}^I(T,U)}$$

$$apply\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : \mathsf{Circ}^I(T,U) \qquad \Theta;\Phi,\Gamma_2;Q_2 \vdash_v W : T}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c \mathsf{apply}(V,W) : U;I}$$

$$box\ \frac{\Theta;\Phi;\emptyset \vdash_v V : !(T \multimap_{I,J} U)}{\Theta;\Phi;\emptyset \vdash_c \mathsf{box}_T\,V : \mathsf{Circ}^I(T,U);0} \qquad nil\ \frac{\Theta \vdash \Phi \qquad \Theta \vdash A}{\Theta;\Phi;\emptyset \vdash_v \mathsf{nil} : \mathsf{List}^0\,A}$$

$$cons\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : A \qquad \Theta;\Phi,\Gamma_2;Q_2 \vdash_v W : \mathsf{List}^I\,A}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1Q_2 \vdash_v \mathsf{cons}\ V\ W : \mathsf{List}^{I+1}\,A}$$

$$fold\ \frac{\begin{array}{c}\Theta;\Phi,\Gamma;Q \vdash_v W : B\{0/i\} \qquad \Theta,i;\Phi;\emptyset \vdash_v V : !((B \otimes A) \multimap_{J,J'} B\{i+1/i\}) \\ \Theta \vdash I \qquad \Theta \vdash A \qquad E = \mathsf{max}(\#(\Gamma;Q), \mathsf{max}_{i<I}\,J + (I-1-i) \times \#(A))\end{array}}{\Theta;\Phi,\Gamma;Q \vdash_v \mathsf{fold}_i\ V\ W : \mathsf{List}^I\,A \multimap_{E,\#(\Gamma;Q)} B\{I/i\}}$$

$$dest\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : A \otimes B \qquad \Theta;\Phi,\Gamma_2,x : A,y : B;Q_2 \vdash_c M : C;I}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c \mathsf{let}\ \langle x,y\rangle = V\ \mathsf{in}\ M : C;I}$$

$$pair\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : A \qquad \Theta;\Phi,\Gamma_2;Q_2 \vdash_v W : B}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_v \langle V,W\rangle : A \otimes B}$$

$$return\ \frac{\Theta;\Gamma;Q \vdash_v V : A}{\Theta;\Gamma;Q \vdash_c \mathsf{return}\ V : A;\#(\Gamma;Q)}$$

$$let\ \frac{\Theta;\Phi,\Gamma_1;Q_1 \vdash_c M : A;I \qquad \Theta;\Phi,\Gamma_2,x : A;Q_2 \vdash_c N : B;J}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c \mathsf{let}\ x = M\ \mathsf{in}\ N : B;\mathsf{max}(I + \#(\Gamma_2;Q_2), J)}$$

$$vsub\ \frac{\Theta;\Gamma;Q \vdash_v V : A \qquad \Theta \vdash_s A <: B}{\Theta;\Gamma;Q \vdash_v V : B}$$

$$csub\ \frac{\Theta;\Gamma;Q \vdash_c M : A;I \qquad \Theta \vdash_s A <: B \qquad \Theta \vDash I \leq J}{\Theta;\Gamma;Q \vdash_c M : B;J}$$

Fig. 12. Proto-Quipper-R type system

the typing context be devoid of linear variables. This reflects the idea that V is meant to be executed in complete isolation, to build a standalone, replicable circuit, and therefore it should not capture any linear resource (e.g. a label) from the surrounding environment.

Wire Count Notice that many rules rely on an operator written $\#(\cdot)$, which we call the *wire count* operator. Intuitively, this operator returns the number of wire resources (in our case, bits or qubits) represented by a type or context. To understand how this is important, consider the *return* rule. The return operator turns a value V into a trivial computation that evaluates immediately to V, and therefore it would be tempting to give it an effect annotation of 0. However, V is not necessarily a closed value. In fact, it might very well contain many bits and qubits, coming both from the typing context Γ and the label context Q. Although nothing happens to these bits and qubits, they still corresponds to wires in the underlying circuit, and these wires have a width which must be accounted for in the judgment for the otherwise trivial computation. The *return* rule therefore produces an effect annotation of the form $\#(\Gamma; Q)$, which is shorthand for $\#(\Gamma)+\#(Q)$ and corresponds exactly to this quantity. A formal definition of the wire count operator on types is given in the following definition, which is lifted to contexts in the natural way.

Definition 2 (Wire Count). *We define the* wire count *of a type A, written $\#(A)$, as a function $\#(\cdot) : TYPE \to INDEX$*

$$\#(\mathbb{1}) = \#(!\,A) = \#(\mathsf{Circ}^I(T,U)) = 0 \qquad\qquad \#(w) = 1$$

$$\#(A \otimes B) = \#(A) + \#(B) \qquad \#(A \multimap_{I,J} B) = J \qquad \#(\mathsf{List}^I\, A) = I \times \#(A)$$

This definition is fairly straightforward, except for the arrow case. By itself, an arrow type does not give us any information about the amount of qubits or bits captured in the corresponding closure. This is precisely where the second index J, which keeps track exactly of this quantity, comes into play. This annotation is introduced by the *abs* rule and allows our analysis to circumvent data hiding.

The *let* rule is another rule in which wire counts are essential. The two terms M and N in $\mathsf{let}\ x = M\ \mathsf{in}\ N$ build the circuits $\mathcal{C}_M$ and $\mathcal{C}_N$, whose widths are bounded by I and J, respectively. Once again, it might be tempting to conclude that the overall circuit built by the let construct has width bounded by $\mathsf{max}(I,J)$, but this fails to take into account the fact that while M is building $\mathcal{C}_M$ starting from the wires contained in Γ_1 and Q_1, we must keep aside the wires contained in Γ_2 and Q_2, which will be used by N to build $\mathcal{C}_N$. These wires must flow alongside $\mathcal{C}_M$ and their width, i.e. $\#(\Gamma_2; Q_2)$, adds up to the total width of the left-hand side of the let construct, leading to an overall width upper bound of $\mathsf{max}(I + \#(\Gamma_2; Q_2), J)$. This situation is better illustrated in Figure 13.

The last rule that makes substantial use of wire counts is *fold*, arguably the most complex rule of the system. The main ingredient of the fold rule is the bound index variable i, which occurs in the accumulator type B and is used to

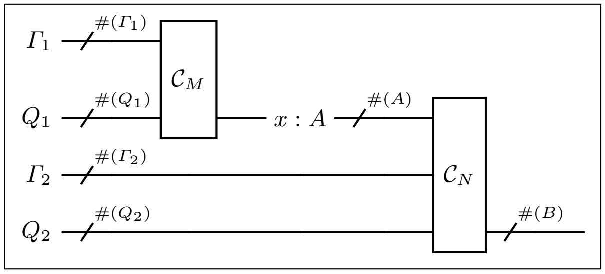

Fig. 13. The shape of a circuit built by a let construct.

keep track of the number of steps performed by the fold. Let $(\cdot)\{I/i\}$ denote the capture-avoiding substitution of the index term I for the index variable i inside an index, type, context, value or term, not unlike $(\cdot)[V/x]$ denotes the capture-avoiding substitution of the value V for the variable x. Intuitively, if the accumulator has initially type $B\{0/i\}$ and each application of the step function increases i by one, then when we fold over a list of length I we get an output of type $B\{I/i\}$. Index E is the upper bound to the width of the overall circuit built by the fold: if the input list is empty, then the width of the circuit is just the number of wires contained in the initial accumulator, that is, $\#(\Gamma; Q)$. If the input list is non-empty, on the other hand, things get slightly more complicated. At each step i, the step function builds a circuit $\mathcal{C}_i$ of width bounded by J, where J might depend on i. This circuit takes as input all the wires in the accumulator, as well as the wires contained in the first element of the input list, which are $\#(A)$. The wires contained in remaining $I-1-i$ elements have to flow alongside $\mathcal{C}_i$, giving a width upper bound of $J + (I-1-i) \times \#(A)$ at each step i. The overall width upper bound is then the maximum for i going from 0 to $I-1$ of this quantity, i.e. precisely $\mathsf{max}_{i<I}\, J + (I-1-i) \times \#(A)$. Once again, a graphical representation of this scenario is given in Figure 14.

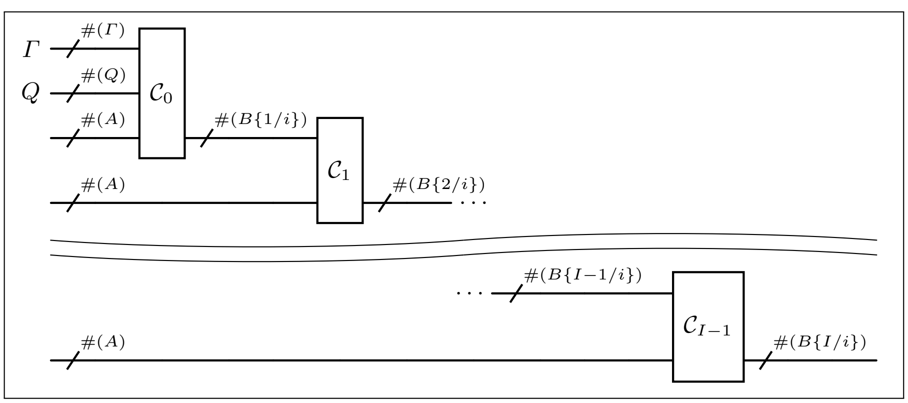

Fig. 14. The shape of a circuit built by a fold applied to an input list of type $\mathsf{List}^I\, A$.

Subtyping Notice that Proto-Quipper-R's type system includes two rules for subtyping, which are effectively the same rule for terms and values, respectively: *csub* and *vsub*. We mentioned that our type system resembles a refinement type system, and all such systems induce a subtyping relation between types, where A is a subtype of B whenever the former is "at least as refined" as the latter. In our case, a subtyping judgment such as $\Theta \vdash_s A <: B$ means that for all natural values of the index variables in Θ, A is a subtype of B.

$$\textit{unit}\,\frac{}{\Theta \vdash_s \mathbb{1} <: \mathbb{1}} \qquad \textit{wire}\,\frac{}{\Theta \vdash_s w <: w} \qquad \textit{bang}\,\frac{\Theta \vdash_s A <: B}{\Theta \vdash_s\, !A <: \,!B}$$

$$\textit{tensor}\,\frac{\Theta \vdash_s A_1 <: A_2 \qquad \Theta \vdash_s B_1 <: B_2}{\Theta \vdash_s A_1 \otimes B_1 <: A_2 \otimes B_2}$$

$$\textit{arrow}\,\frac{\Theta \vdash_s A_2 <: A_1 \qquad \Theta \vdash_s B_1 <: B_2 \qquad \Theta \vDash I_1 \leq I_2 \qquad \Theta \vDash J_1 = J_2}{\Theta \vdash_s A_1 \multimap_{I_1,J_1} B_1 <: A_2 \multimap_{I_2,J_2} B_2}$$

$$\textit{list}\,\frac{\Theta \vdash_s A <: B \qquad \Theta \vDash I = J}{\Theta \vdash_s \mathsf{List}^I\, A <: \mathsf{List}^J\, B}$$

$$\textit{circ}\,\frac{\Theta \vdash_s T_1 <:> T_2 \qquad \Theta \vdash_s U_1 <:> U_2 \qquad \Theta \vDash I \leq J}{\Theta \vdash_s \mathsf{Circ}^I(T_1, U_1) <: \mathsf{Circ}^J(T_2, U_2)}$$

Fig. 15. Proto-Quipper-R subtyping rules

We derive this kind of judgments by the rules in Figure 15. Note that $\Theta \vdash_s A <:> B$ is shorthand for "$\Theta \vdash_s A <: B$ and $\Theta \vdash_s B <: A$". Subtyping relies in turn on a judgment of the form $\Theta \vDash I \leq J$, which is a generalization of the semantic judgment that we used in the CRL type system in Section 4.2. Such a judgment asserts that for all natural values of the index variables in Θ, I is lesser or equal than J. Consequently, $\Theta \vDash I = J$ is shorthand for "$\Theta \vDash I \leq J$ and $\Theta \vDash J \leq I$". We purposefully leave the decision procedure of this kind of judgments unspecified, with the prospect that, from a more practical perspective, they could be delegated to an SMT solver [7].

4.4 Operational Semantics

Operationally speaking, it does not make sense, in the Proto-Quipper languages, to speak of the semantics of a term *in isolation*: a term is always evaluated in the context of an underlying circuit that supplies all of the term's free labels. We therefore define the operational semantics of Proto-Quipper-R as a big-step evaluation relation $\Downarrow$ on *configurations*, i.e. circuits paired with either terms or values. Intuitively, $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$ means that M evaluates to V and updates $\mathcal{C}$ to $\mathcal{D}$ as a side effect.

The rules for evaluating configurations are given in Figure 16, where $\mathcal{C}, \mathcal{D}$ and $\mathcal{E}$ are circuits, M and N are terms, while V, W, X, Y and Z are values. Most

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

$$force\ \frac{(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)}{(\mathcal{C}, \mathsf{force}(\mathsf{lift}\, M)) \Downarrow (\mathcal{D}, V)} \qquad apply\ \frac{(\mathcal{E}, \bar{q}) = \mathrm{append}(\mathcal{C}, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))}{(\mathcal{C}, \mathsf{apply}((\bar{\ell}, \mathcal{D}, \bar{k}), \bar{t})) \Downarrow (\mathcal{E}, \bar{q})}$$

$$box\ \frac{(Q, \bar{\ell}) = \mathrm{freshlabels}(T) \qquad (id_Q, M) \Downarrow (id_Q, V) \qquad (id_Q, V\, \bar{\ell}) \Downarrow (\mathcal{D}, \bar{k})}{(\mathcal{C}, \mathsf{box}_T(\mathsf{lift}\, M)) \Downarrow (\mathcal{C}, (\bar{\ell}, \mathcal{D}, \bar{k}))}$$

$$return\ \frac{}{(\mathcal{C}, \mathsf{return}\ V) \Downarrow (\mathcal{C}, V)} \qquad let\ \frac{(\mathcal{C}, M) \Downarrow (\mathcal{E}, V) \qquad (\mathcal{E}, N[V/x]) \Downarrow (\mathcal{D}, W)}{(\mathcal{C}, \mathsf{let}\ x = M\ \mathsf{in}\ N) \Downarrow (\mathcal{D}, W)}$$

$$fold\text{-}end\ \frac{}{(\mathcal{C}, (\mathsf{fold}_i\ V\ W)\, \mathsf{nil}) \Downarrow (\mathcal{C}, W)}$$

$$fold\text{-}step\ \frac{\begin{array}{c}(\mathcal{C}, M\{0/i\}) \Downarrow (\mathcal{C}, Y) \qquad (\mathcal{C}, Y\, \langle V, W\rangle) \Downarrow (\mathcal{E}, Z)\\ (\mathcal{E}, (\mathsf{fold}_i\ (\mathsf{lift}\, M\{i+1/i\})\ Z)\, W') \Downarrow (\mathcal{D}, X)\end{array}}{(\mathcal{C}, (\mathsf{fold}_i\ (\mathsf{lift}\, M)\ V)\, (\mathsf{cons}\, W\ W')) \Downarrow (\mathcal{D}, X)}$$

Fig. 16. Proto-Quipper-R big-step operational semantics.

evaluation rules are straightforward, with the exception perhaps of *apply, box* and *fold-step*. Being the fundamental block of circuit-building, the semantics of apply lies almost entirely in the way it updates the underlying circuit. The concatenation of the underlying circuit $\mathcal{C}$ and the applicand $\mathcal{D}$ is delegated entirely to the append function, which is defined as follows:

Definition 3 (append**).** *We define* the append of $(\bar{\ell}, \mathcal{D}, \bar{k})$ to $\mathcal{C}$ on $\bar{t}$, *written* $\mathrm{append}(\mathcal{C}, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))$, *as the function that performs the following steps:*

1. *Finds* $(\bar{t}, \mathcal{D}', \bar{q})$ *equivalent to* $(\bar{\ell}, \mathcal{D}, \bar{k})$ *such that the labels shared by* $\mathcal{C}$ *and* $\mathcal{D}'$ *are all and only those in* $\bar{t}$*,*
2. *Computes* $\mathcal{E} = \mathcal{C} :: \mathcal{D}'$*,*
3. *Returns* $(\mathcal{E}, \bar{q})$*.*

Note that two circuits are *equivalent* when they only differ by a renaming of labels. What the renaming does, in this case, is instantiate the generic input interface $\bar{\ell}$ of circuit $\mathcal{D}$ with the actual labels that it is going to be appended to, namely $\bar{t}$, and ensure that there are no name clashes between the labels occurring in the resulting $\mathcal{D}'$ and those occurring in $\mathcal{C}$.

On the other hand, the semantics of a term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$ relies on the freshlabels function. What freshlabels does is take as input a bundle type T and instantiate fresh $Q, \bar{\ell}$ such that $Q \vdash_w \bar{\ell} : T$. The wire bundle $\bar{\ell}$ is then used as a dummy argument to V, the circuit-building function resulting from the evaluation of M. This function application is evaluated in the context of the identity circuit id_Q and eventually produces a circuit $\mathcal{D}$, together with its output labels $\bar{k}$. Finally, $\bar{\ell}$ and $\bar{k}$ become respectively the input and output interfaces of the boxed circuit $(\bar{\ell}, \mathcal{D}, \bar{k})$, which is the result of the evaluation of $\mathsf{box}_T(\mathsf{lift}\, M)$.

Note at this point that T controls how many labels are initialized by the freshlabels function. Because T can contain indices (e.g. it could be that $T \equiv \mathsf{List}^3\,\mathsf{Qubit}$), it follows that in Proto-Quipper-R indices are not only relevant to typing, but they also have operational value. For this reason, the semantics of Proto-Quipper-R is well-defined only on terms closed both in the sense of regular variables *and* index variables, since a circuit-building function of input type, say, $\mathsf{List}^i\,\mathsf{Qubit}$ does not correspond to any individual circuit, and therefore it makes no sense to try and box it. This aspect of the semantics is also apparent in the *fold-step* rule, where the index variable i occurring free in M is instantiated to 0 before evaluating M to obtain the step function Y. Then, before evaluating the next fold, i is replaced with $i+1$ in M, increasing the index by one for the next iteration.

5 Type Safety and Correctness

Because the operational semantics of Proto-Quipper-R is based on configurations, we ought to adopt a notion of well-typedness which is also based on configurations. The following definition of *well-typed configuration* is thus central to our type-safety analysis.

Definition 4 (Well-typed Configuration). *We say that configuration $(\mathcal{C}, M)$ is* well-typed with input Q, type A, width I and output L, *and we write $Q \vdash (\mathcal{C}, M) : A; I; L$, whenever $\mathcal{C} : Q \to L, H$ for some H such that $\emptyset; \emptyset; H \vdash_c M : A; I$. We write $Q \vdash (\mathcal{C}, V) : A; L$ whenever $\mathcal{C} : Q \to L, H$ for some H such that $\emptyset; \emptyset; H \vdash_v V : A$.*

The three results that we want to show in this section are that any well-typed term configuration $Q \vdash (\mathcal{C}, M) : A; I; L$ evaluates to some configuration $(\mathcal{D}, V)$, that $Q \vdash (\mathcal{D}, V) : A; L$ and that $\mathcal{D}$ is obtained from $\mathcal{C}$ by extending it with a sub-circuit of width at most I. These claims correspond to the *subject reduction* and *total correctness* properties that we will prove at the end of this section. However, both these results rely on a central lemma and on the mutual notions of *realization* and *reducibility*, which we first give formally.

Definition 5 (Realization). *We define $V \Vdash_Q A$, which reads V* realizes A under Q, *as the smallest relation such that*

- $* \Vdash_\emptyset \mathbb{1}$,
- $\ell \Vdash_{\ell:w} w$,
- $V \Vdash_Q A \multimap_{I,J} B$ *iff* $\vDash J = |Q|$ *and* $\forall W : W \Vdash_L A \implies V\,W \Vdash^I_{Q,L} B$,
- $\mathsf{lift}\,M \Vdash_\emptyset\, !A$ *iff* $M \Vdash^0_\emptyset A$,
- $\langle V, W\rangle \Vdash_{Q,L} A \otimes B$ *iff* $V \Vdash_Q A$ *and* $W \Vdash_L B$,
- $\mathsf{nil} \Vdash_\emptyset \mathsf{List}^I A$ *iff* $\vDash I = 0$,
- $\mathsf{cons}\,V\,W \Vdash_{Q,L} \mathsf{List}^I A$ *iff* $\vDash I = J + 1$ *and* $V \Vdash_Q A$ *and* $W \Vdash_L \mathsf{List}^J A$,
- $(\bar{\ell}, \mathcal{C}, \bar{k}) \Vdash_\emptyset \mathsf{Circ}^I(T, U)$ *iff* $\mathcal{C} : Q \to L$ *and* $Q \vdash_w \bar{\ell} : T$ *and* $L \vdash_w \bar{k} : U$ *and* $\vDash \mathrm{width}(\mathcal{C}) \leq I$.

Definition 6 (Reducibility). *We say that M* is reducible under Q with type A and width I, *and we write $M \Vdash^I_Q A$, if, for all $\mathcal{C}$ such that $\mathcal{C} : L \to Q, H$, there exist $\mathcal{D}, V$ such that*

1. $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$,
2. $\vDash \mathrm{width}(\mathcal{D}) \leq I$
3. $\mathcal{D} : Q \to K$ *for some K such that $V \Vdash_K A$.*

Both relations, and in particular reducibility, are given in the form of unary logical relations [55]. The intuition is pretty straightforward: a term is reducible with width I if it evaluates correctly when paired with any circuit $\mathcal{C}$ which provides its free labels and if it extends $\mathcal{C}$ with a sub-circuit $\mathcal{D}$ whose width is bounded by I. Realization, on the other hand, is less immediate. For most cases, realizing type A loosely corresponds to being closed and well-typed with type A, but a value realizes an arrow type $A \multimap_{I,J} B$ when its application to a value realizing A is reducible with type B and width I.

By themselves, realization and reducibility are defined only on terms and values closed in the sense both of regular and index variables. To extend these notions to open terms and values, we adopt the standard approach of reasoning explicitly about the substitutions that would render them closed. A *closing value substitution* γ is a function that turns an open term M into a closed term $\gamma(M)$ by substituting a value for each free variable occurring in M. We say that γ *implements* a typing context Γ using label context Q, and we write $\gamma \vDash_Q \Gamma$, when it replaces every variable x_i in the domain of Γ with a value V_i such that $V_i \Vdash_{Q_i} \Gamma(x_i)$ and $Q = \biguplus_{x_i \in \mathrm{dom}(\Gamma)} Q_i$. A *closing index substitution* θ is similar, only it substitutes closed indices for index variables and can be applied to indices, types, contexts, values and terms alike. We say that θ implements an index context Θ, and we write $\theta \vDash \Theta$, when it replaces every index variable in Θ with a closed index term. This allows us to give the following fundamental lemma, which will be used while proving all other claims.

Lemma 1 (Core Correctness). *Let Π be a type derivation. For all $\theta \vDash \Theta$ and $\gamma \vDash_Q \theta(\Gamma)$, we have that*

$$\Pi \triangleright \Theta; \Gamma; L \vdash_c M : A; I \implies \gamma(\theta(M)) \Vdash^{\theta(I)}_{Q,L} \theta(A)$$
$$\Pi \triangleright \Theta; \Gamma; L \vdash_v V : A \implies \gamma(\theta(V)) \Vdash_{Q,L} \theta(A)$$

Proof. By induction on the size of Π, making use of Theorem 1.

Lemma 1 tells us that any well-typed term (resp. value) is reducible (resp. realizes its type) when we instantiate its free variables according to its contexts. Now that we have Lemma 1, we can proceed to proving the aforementioned results of subject reduction and total correctness. We start with the former, which unsurprisingly requires the following substitution lemmata.

Lemma 2 (Index Substitution). *Let Π be a type derivation and let I be an index such that $\Theta \vdash I$. We have that*

$$\Pi \triangleright \Theta, i; \Gamma; Q \vdash_c M : A; J \implies \Theta; \Gamma\{I/i\}; Q \vdash_c M\{I/i\} : A\{I/i\}; J\{I/i\},$$
$$\Pi \triangleright \Theta, i; \Gamma; Q \vdash_v V : A \implies \Theta; \Gamma\{I/i\}; Q \vdash_v V\{I/i\} : A\{I/i\}.$$

Proof. By induction on the size of Π.

Lemma 3 (Value Substitution). *Let Π be a type derivation and let V be a value such that $\Theta;\Phi,\Gamma_1;Q_1 \vdash_v V : A$. We have that*

$$\Pi \triangleright \Theta;\Phi,\Gamma_2,x:A;Q_2 \vdash_c M : B;I \implies \Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c M[V/x] : B;I,$$
$$\Pi \triangleright \Theta;\Phi,\Gamma_2,x:A;Q_2 \vdash_v W : B \implies \Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_v W[V/x] : B.$$

Proof. By induction on the size of Π.

Theorem 2 (Subject Reduction). *If $Q \vdash (\mathcal{C}, M) : A;I;L$ and $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$, then $Q \vdash (\mathcal{D}, V) : A;L$.*

Proof. By induction on the derivation of $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$ and case analysis on the last rule used in its derivation. Lemma 3 is essential to the *app,dest* and *let* cases, while Lemma 2 is used in the *fold-step* case. Lemma 1 is essential to the *box* case, as it is the only case in which the side effect of the evaluation (the circuit built by the function being boxed), whose preservation is the a matter of correctness, becomes a value (the resulting boxed circuit).

Of course, type soundness is not enough: we also want the resource analysis carried out by our type system to be correct, as stated in the following theorem.

Theorem 3 (Total Correctness). *If $Q \vdash (\mathcal{C}, M) : A;I;L$, then there exist $\mathcal{D}, V$ such that $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$ and $\vDash \mathrm{width}(\mathcal{D}) \leq I$.*

Proof. By definition, $Q \vdash (\mathcal{C}, M) : A;I;L$ entails that $\mathcal{C} : Q \to L, H$ and $\emptyset;\emptyset;H \vdash_c M : A;I$. Since an empty context is trivially implemented by an empty closing substitution, by Lemma 1 we get $M \Vdash_H^I A$, which by definition entails that there exist $\mathcal{D}, V$ such that $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$ and $\vDash \mathrm{width}(\mathcal{D}) \leq I$.

6 A Practical Example

This section provides an example of how Proto-Quipper-R can be used to verify the resource usage of realistic quantum algorithms. In particular, we use our language to implement the QFT algorithm [12, 40] and verify that the circuits it produces have width no greater than the size of their input, i.e. that the QFT algorithm does not overall use additional ancillary qubits.

The Proto-Quipper-R implementation of the QFT algorithm is given in Figure 17. As we walk through the various parts of the program, be aware that we will focus on the resource aspects of the algorithm, ignoring much of its actual meaning. Starting bottom-up, we assume that we have an encoding of naturals in the language and that we can perform arithmetic on them. We also assume some primitive gates and gate families: H is the boxed circuit corresponding to the Hadamard gate and has type $\mathsf{Circ}^1(\mathsf{Qubit}, \mathsf{Qubit})$, whereas the $\mathsf{makeRGate}$ function has type $\mathsf{Nat} \multimap_{0,0} \mathsf{Circ}^2(\mathsf{Qubit} \otimes \mathsf{Qubit}, \mathsf{Qubit} \otimes \mathsf{Qubit})$ and produces instances of the parametric controlled R_n gate. On the other hand, *qlen* and

$$
\begin{aligned}
&\mathit{qft} \triangleq \mathsf{fold}_j\ \mathit{qftStep}\ \mathsf{nil}\\
&\mathit{qftStep} \triangleq \mathsf{lift}(\mathsf{return}\ \lambda\langle qs, q\rangle_{\mathsf{List}^j\ \mathsf{Qubit}\otimes\mathsf{Qubit}}.\\
&\qquad \mathsf{let}\ \langle n, qs\rangle = \mathit{qlen}\ qs\ \mathsf{in}\\
&\qquad \mathsf{let}\ \mathit{revQs} = \mathit{rev}\ qs\ \mathsf{in}\\
&\qquad \mathsf{let}\ \langle q, qs\rangle = (\mathsf{fold}_e\ (\mathsf{lift}(\mathit{rotate}\, n))\ \langle q, \mathsf{nil}\rangle)\ \mathit{revQs}\ \mathsf{in}\\
&\qquad \mathsf{let}\ q = \mathsf{apply}(\mathsf{H}, q)\ \mathsf{in}\\
&\qquad \mathsf{return}\ (\mathsf{cons}\ q\ qs))\\
&\\
&\mathit{rotate} \triangleq \lambda n_{\mathsf{Nat}}.\mathsf{return}\ \lambda\langle\langle q, cs\rangle, c\rangle_{(\mathsf{Qubit}\otimes\mathsf{List}^e\ \mathsf{Qubit})\otimes\mathsf{Qubit}}.\\
&\qquad \mathsf{let}\ \langle m, cs\rangle = \mathit{qlen}\ cs\ \mathsf{in}\\
&\qquad \mathsf{let}\ \mathit{rgate} = \mathsf{makeRGate}\,(n + 1 - m)\ \mathsf{in}\\
&\qquad \mathsf{let}\ \langle q, c\rangle = \mathsf{apply}(\mathit{rgate}, \langle q, c\rangle)\ \mathsf{in}\\
&\qquad \mathsf{return}\ \langle q, \mathsf{cons}\ c\ cs\rangle
\end{aligned}
$$

Fig. 17. A Proto-Quipper-R implementation of the Quantum Fourier Transform circuit family. The usual syntactic sugar is employed.

rev stand for regular language terms which implement respectively the linear list length and reverse functions. They have type $\mathit{qlen} :: \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} (\mathsf{Nat} \otimes \mathsf{List}^i\ \mathsf{Qubit})$ and $\mathit{rev} :: \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\ \mathsf{Qubit}$ in our type system.

We now turn our attention to the actual QFT algorithm. Function *qftStep* builds a single step of the QFT circuit. The width of the circuit produced at step j is dominated by the folding of the $\mathit{rotate}\, n$ function, which applies controlled rotations between appropriate pairs of qubits and has type

$$(\mathsf{Qubit} \otimes \mathsf{List}^e\ \mathsf{Qubit}) \otimes \mathsf{Qubit} \multimap_{e+2,0} \mathsf{Qubit} \otimes \mathsf{List}^{e+1}\ \mathsf{Qubit}, \tag{5}$$

meaning that $\mathit{rotate}\, n$ rearranges the structure of its inputs, but overall does not introduce any new wire. We fold this function starting from an accumulator $\langle q, \mathsf{nil}\rangle$, meaning that we can give $\mathsf{fold}_j\ (\mathsf{lift}(\mathit{rotate}\, n))\ \langle q, \mathsf{nil}\rangle$ type as follows:

$$\mathit{fold}\,\frac{\begin{array}{c} i, j, e; n : \mathsf{Nat}; \emptyset \vdash_v \mathsf{lift}(\mathit{rotate}\, n) : !((\mathsf{Q} \otimes \mathsf{List}^e\ \mathsf{Q}) \otimes \mathsf{Q} \multimap_{e+2,0} \mathsf{Q} \otimes \mathsf{List}^{e+1}\ \mathsf{Q}) \\ i, j; q : \mathsf{Q}; \emptyset \vdash_v \langle q, \mathsf{nil}\rangle : \mathsf{Q} \otimes \mathsf{List}^0\ \mathsf{Q} \qquad i, j \vdash j \qquad i, j \vdash \mathsf{Q} \end{array}}{i, j; n : \mathsf{Nat}, q : \mathsf{Q}; \emptyset \vdash_v \mathsf{fold}_e\ \mathsf{lift}(\mathit{rotate}\, n)\ \langle q, \mathsf{nil}\rangle : \mathsf{List}^j\ \mathsf{Q} \multimap_{j+1,1} \mathsf{Q} \otimes \mathsf{List}^j\ \mathsf{Q}} \tag{6}$$

where Q is shorthand for Qubit and where we implicitly use the fact that $i, j \vDash \mathsf{max}(1, \mathsf{max}_{e<j}\, e + 2 + (j - 1 - e) \times 1) = j + 1$ to simplify the arrow's width annotation using *vsub* and the *arrow* subtyping rule. Next, we fold over *revQs*, which has the same elements as qs and thus has length j, and we obtain that the fold produces a circuit whose width is bounded by $j + 1$. Therefore, *qftStep* has type

$$!((\mathsf{List}^j\ \mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\ \mathsf{Qubit}), \tag{7}$$

which entails that when we pass it as an argument to the topmost fold together with nil we can conclude that the type of the *qft* function is

$$\mathit{fold}\,\frac{\begin{array}{c} i, j; \emptyset; \emptyset \vdash_v \mathit{qftStep} : !((\mathsf{List}^j\ \mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\ \mathsf{Qubit}) \\ i; \emptyset; \emptyset \vdash_v \mathsf{nil} : \mathsf{List}^0\ \mathsf{Qubit} \qquad\qquad i \vdash i \qquad\qquad i \vdash \mathsf{Qubit} \end{array}}{i; \emptyset; \emptyset \vdash_v \mathsf{fold}_j\ \mathit{qftStep}\ \mathsf{nil} : \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\ \mathsf{Qubit}} \tag{8}$$

where we once again implicitly simplify the arrow type using the fact that $i \vDash \mathsf{max}(0, \mathsf{max}_{j<i}\, j + 1 + (i - 1 - j) \times 1) = i$. This concludes our analysis and the resulting type tells us that *qft* produces a circuit of width at most i on inputs of size i, without overall using any additional wires. If we instantiate i to 3, for example, we can apply *qft* to a list of 3 qubits to obtain the circuit shown in Figure 18, whose width is exactly 3.

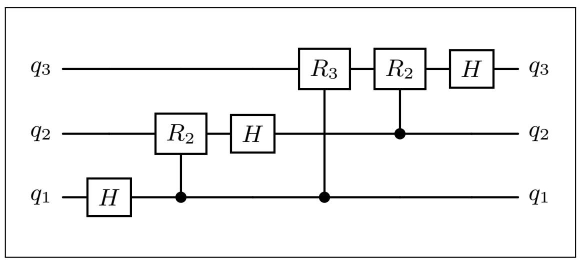

Fig. 18. The circuit of input size 3 produced by *qft* ($\mathsf{cons}\ q_1\ \mathsf{cons}\ q_2\ \mathsf{cons}\ q_3\ \mathsf{nil}$)

To conclude this section, note that for ease of exposition *qft* actually produces the *reversed* QFT circuit. This is not a problem, since the two circuits are equivalent resource-wise and the actual QFT circuit can be recovered by boxing the result of *qft* and reversing it via a primitive operator. Besides, note that `Quipper`'s internal implementation of the QFT is also reversed [17].

7 Related Work

The metatheory of quantum circuit description languages, and in particular of `Quipper`-style languages, has been the subject of quite some work in recent years, starting with Ross's thesis on Proto-Quipper-S [48] and going forward with Selinger and Rios's Proto-Quipper-M [46]. In the last five years, some proposals have also appeared for more expressive type systems or for language extensions that can handle non-standard language features, such as the so-called *dynamic lifting* [36, 22, 9], available in the `Quipper` language, or dependent types [23]. Although some embryonic contributions in the direction of analyzing the size of circuits produced using `Quipper` have been given [56], no contribution tackles the problem of deriving resource bounds *parametric* on the size of the input. In this, the ability to have types which depend on the input, certainly a feature of Proto-Quipper-D [23], is not useful for the analysis of intensional attributes of the underlying circuit, simply because such attributes are not visible in types.

If we broaden the horizon to quantum programming languages other than `Quipper`, we come across, for example, the recent works of Avanzini et al. [5] and Liu et al. [37] on adapting the classic weakest precondition technique to the cost analysis of quantum programs, which however focus on programs in an imperative language. The work of Dal Lago et al. [14] on a quantum language which characterizes complexity classes for quantum polynomial time should certainly be remembered: even though the language allows the use of higher order functions, the manipulation of quantum data occurs directly and not through circuits. Similar considerations hold for the recent work of Hainry et al. [30] and Yamakami's algebra of functions [59] in the style of Bellantoni and Cook [6], both characterizing quantum polynomial time.

If we broaden our scope further and become interested in the analysis of the cost of classical or probabilistic programs, we face a vast literature, with contributions employing a variety of techniques on heterogeneous languages and calculi: from functional programs [2, 34, 33] and term rewriting systems [3, 4, 42] to probabilistic [35] and object-oriented programs [29, 20]. In this context, the resource under analysis is often assumed to be computation *time*, which is relatively easy to analyze given its strictly monotonic nature. Circuit width, although monotonically non-decreasing, evolves in a way that depends on a non-monotonic quantity, i.e. the number of wires discarded by a circuit. As a result, width has the flavor of space and its analysis is less straightforward.

It is also worth mentioning that linear dependent types can be seen as a specialized version of refinement types [19], which have been used extensively in the literature to automatically verify interesting properties of programs [60, 38]. In particular, the work of Vazou et al. on `Liquid Haskell` [58, 57] has been of particular inspiration, on account of `Quipper` being embedded precisely in `Haskell`. The liquid type system [47] of `Liquid Haskell` relies on SMT solvers to discharge proof obligations and has been used fruitfully to reason about both the correctness and the resource consumption (mainly time complexity) of concrete, practical programs [31].

8 Generalization to Other Resource Types

This work focuses on estimating the *width* of the circuits produced by `Quipper` programs. This choice is dictated by the fact that the width of a circuit corresponds to the maximum number of distinct wires, and therefore individual qubits, required to execute it. Nowadays, this is considered as one of the most precious resources in quantum computing, and as such must be kept under control. However, this does not mean that our system could not be adapted to the estimation of other parameters. This section outlines how this may be possible.

First, estimating strictly monotonic resources, such as the total *number of gates* in a circuit, is possible and in fact simpler than estimating width. A *sin-*

gle index term I that measures the number of gates in the circuit built by a computation would be enough to carry out this analysis. This index would be appropriately increased any time an apply instruction is executed, while sequencing two terms via let would simply add together the respective indices.

If the parameter of interest were instead the *depth* of the circuit, then the approach would have to be slightly different. Although in principle it would be possible to still rely on a single index I, this would give rise to a very coarse approximation, effectively collapsing the analysis of depth to a gate count analysis. A more precise approximation could instead be obtained by keeping track of depth *locally* rather than *globally*. More specifically, it would be sufficient to decorate each occurrence of a wire type w with an index term I so that if a label ℓ were typed with w^I, it would mean that the sub-circuit rooted in ℓ has a depth at most equal to I.

Finally, it should be mentioned that the resources considered, i.e. the depth, width, and gate count of a circuit, can be further refined so as to take into account only *some* kinds of wires and gates. For instance, one could want to keep track of the maximum number of *qubits* needed, ignoring the number of classical bits, or at least distinguishing the two parameters, which of course have distinct levels of criticality in current quantum hardware.

9 Conclusion and Future Work

In this paper we introduced a linear dependent type system based on index refinements and effect typing for the paradigmatic calculus Proto-Quipper, with the purpose of using it to derive upper bounds on the width of the circuits produced by programs. We proved not only the classic type safety properties, but also that the upper bounds derived via the system are correct. We also showed how our system can verify a realistic quantum algorithm and elaborated on some ideas on how our technique could be adapted to other crucial resources types, like gate count and circuit depth. Ours is the first type system designed specifically for the purpose of resource analysis to target circuit description languages such as `Quipper`. Technically, the main novelties are the smooth combination of effect typing and index refinements, but also the proof of correctness, in which reducibility and effects are shown to play well together.

Among topics for further work, we can identify three main research directions. First and foremost, it would be valuable to investigate the ideas presented in this paper from a more practical perspective, that is, to provide a prototype implementation of the language and, more importantly, of the type-checking procedure. This would require understanding the role that SMT solvers may play in discharging the semantic judgments which we use pervasively in our approach.

Staying instead on the theoretical side of things, on one hand we have the prospect of denotational semantics: most incarnations of Proto-Quipper are endowed with categorical semantics that model both circuits and the terms of the language that build them [46, 36, 23, 22]. We already mentioned how the intensional nature of the quantity under analysis renders the formulation of an abstract categorical semantics for Proto-Quipper-R and its circuits a nontrivial task, but we believe that one such semantics would help Proto-Quipper-R fit better in the Proto-Quipper landscape.

On the other hand, in Section 8 we briefly discussed how our system could be modified to handle the analysis of different resource types. It would be interesting to test this path and to investigate the possibility of *actually generalizing* our resource analysis, that is, of making it parametric on the kind of resource being analyzed. This would allow for the same program in the same language to be amenable to different forms of verification, in a very flexible fashion.

References

1. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: Proc. of LICS '05 (2005)
2. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the complexity of functional programs: Higher-order meets first-order. In: Proc. of ICFP 2015 (2015)
3. Avanzini, M., Moser, G.: Complexity analysis by rewriting. In: Proc. of FLOPS 2008 (2008)
4. Avanzini, M., Moser, G.: Tyrolean Complexity Tool: Features and Usage. In: Proc. of RTA 2013. vol. 21 (2013)
5. Avanzini, M., Moser, G., Pechoux, R., Perdrix, S., Zamdzhiev, V.: Quantum expectation transformers for cost analysis. In: Proc. of LICS '22 (2022)
6. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions (extended abstract). In: STOC '92 (1992)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press (2021)
8. Cleve, R., Ekert, A., Macchiavello, C., Mosca, M.: Quantum algorithms revisited. Proc. Math. Phys. Eng. Sci. P ROY SOC A-MATH PHY **454**(1969) (1998)
9. Colledan, A., Dal Lago, U.: On Dynamic Lifting and Effect Typing in Circuit Description Languages. In: TYPES 2022. vol. 269 (2023)
10. Collins, H., Nay, C.: Ibm unveils 400 qubit-plus quantum processor and next-generation ibm quantum system two (2022), https://is.gd/WPV7lO, retrieved on Oct. 15, 2023
11. Conover, E.: Light-based quantum computer jiuzhang achieves quantum supremacy (2020), https://is.gd/kFv6IOy, retrieved on Oct. 15, 2023
12. Coppersmith, D.: An approximate fourier transform useful in quantum factoring, ibm research report rc19642 (2002)
13. Dal Lago, U., Gaboardi, M.: Linear dependent types and relative completeness. In: Proc. of LICS '11 (2011)
14. Dal Lago, U., Masini, A., Zorzi, M.: Quantum implicit computational complexity. TCS **411**(2) (2010)
15. Dal lago, U., Petit, B.: Linear dependent types in a call-by-value scenario. In: Proc. of PPDP '12 (2012)
16. Dal lago, U., Petit, B.: The geometry of types. In: Proc. of POPL '13 (2013)
17. Eisenberg, R., Green, A., Lumsdaine, P., Kim, K., Mau, S.C., Mohan, B., Ng, W., Ravelomanantsoa-Ratsimihah, J., Ross, N., Scherer, A., Selinger, P., Valiron, B., Virodov, A., Zdancewic, S.: Quipper.libraries.qft, https://is.gd/AEJmp9, retrieved on Oct. 15, 2023

18. Fingerhuth, M., Babej, T., Wittek, P.: Open source software in quantum computing. PLOS ONE **13**(12) (2018)
19. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proc. of PLDI '91 (1991)
20. Frohn, F., Giesl, J.: Complexity analysis for java with aprove. In: Proc. of IFM 2017 (2017)
21. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In: Proc. of RC (2020)
22. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-quipper with dynamic lifting. In: Proc. of POPL '23 (2023)
23. Fu, P., Kishida, K., Selinger, P.: Linear dependent type theory for quantum programming languages: Extended abstract. In: Proc. of LICS '20 (2020)
24. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Proc. of POPL '13 (2013)
25. Gay, S.J.: Quantum programming languages: Survey and bibliography. Math. Struct. Comput. Sci. **16**(4) (2006)
26. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Proc. of RC (2013)
27. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper. In: Proc. of PLDI (2013)
28. Grover, L.K.: A fast quantum mechanical algorithm for database search (1996)
29. Hainry, E., Péchoux, R.: Type-based heap and stack space analysis in Java (2013), technical report
30. Hainry, E., Péchoux, R., Silva, M.: A programming language characterizing quantum polynomial time. In: Proc. of FoSSaCS 2023 (2023)
31. Handley, M., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in liquid haskell. PACMPL **4** (2019)
32. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Phys. Rev. Lett. **103** (2009)
33. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ml. In: Computer Aided Verification (2012)
34. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Programming Languages and Systems (2010)
35. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. In: Programming Languages and Systems. Berlin, Heidelberg (2016)
36. Lee, D., Perrelle, V., Valiron, B., Xu, Z.: Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In: Proc. of FSTTCS (2021)
37. Liu, J., Zhou, L., Barthe, G., Ying, M.: Quantum weakest preconditions for reasoning about expected runtimes of quantum programs. In: Proc. of LICS '22 (2022)
38. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proc. of ICFP '03 (2003)
39. Martinis, J.: Quantum supremacy using a programmable superconducting processor (2019), https://is.gd/v3VXFi, retrieved on Oct. 15, 2023
40. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010)
41. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design: Recent Insights and Advances (1999)
42. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. Journal of Automated Reasoning **51**(1) (2013)
43. Palsberg, J.: Toward a universal quantum programming language. XRDS: Crossroads **26**(1) (2019)

44. Paykin, J., Rand, R., Zdancewic, S.: Qwire: A core language for quantum circuits. In: Proc. of POPL '17 (2017)
45. Preskill, J.: Quantum computing and the entanglement frontier (2012)
46. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. In: Proc. of QPL '17 (2017)
47. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proc. of PLDI '08 (2008)
48. Ross, N.: Algebraic and Logical Methods in Quantum Computation. Ph.D. thesis (2015)
49. Sanders, J.W., Zuliani, P.: Quantum programming. In: Proc. of MPC 2000 (2000)
50. Schlosshauer, M.: Decoherence and the Quantum-To-Classical Transition. Springer Berlin Heidelberg (2007)
51. Selinger, P.: A brief survey of quantum programming languages. In: Proc. of FLOPS 2004 (2004)
52. Selinger, P.: Towards a quantum programming language. Math. Struct. Comput. Sci. **14**(4) (2004)
53. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. In: Proc. of TLCA (2005)
54. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proc. of FOCS '94 (1994)
55. Skorstengaard, L.: An introduction to logical relations (2019)
56. Valiron, B.: Automated, parametric gate count of quantum programs (2016), https://is.gd/XIm3lh, retrieved on Oct. 15 2023
57. Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: Experience with refinement types in the real world. In: Proc. of Haskell '14 (2014)
58. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for Haskell. In: Proc. of ICFP '14 (2014)
59. Yamakami, T.: A schematic definition of quantum polynomial time computability. The Journal of Symbolic Logic **85**(4) (2020)
60. Çiçek, E., Garg, D., Acar, U.: Refinement Types for Incremental Computational Complexity. In: Programming Languages and Systems, vol. 9032 (2015)
61. Ömer, B.: Classical concepts in quantum programming. Int. J. Theor. Phys. **44**(7) (2005)