Circuit Width Estimation via Effect Typing and Linear Dependency

ANDREA COLLEDAN and UGO DAL LAGO, University of Bologna, Bologna, Italy and INRIA Sophia Antipolis, Sophia Antipolis, France
NIKI VAZOU, IMDEA Software, Pozuelo de Alarcón, Spain

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 quantum algorithms, whose circuits, however, may involve many more qubits and gate applications than current quantum architectures can actually muster. In this article, 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. Lastly, we introduce QuRA, a static analysis tool based on Proto-Quipper-R's type system, and use it to show that our framework allows for the automatic width verification of realistic quantum algorithms, such as the QFT and Grover's algorithm.

CCS Concepts: • **Theory of computation** → **Program verification**; **Quantum complexity theory**; **Lambda calculus**; **Type theory**; **Operational semantics**; • **Software and its engineering** → **Domain specific languages**; • **Hardware** → **Quantum computation**;

Additional Key Words and Phrases: Effect Typing, Linear Dependent Types, Lambda Calculus, Quantum Computing, Quipper

ACM Reference format:
Andrea Colledan, Ugo Dal Lago, and Niki Vazou. 2025. Circuit Width Estimation via Effect Typing and Linear Dependency. *ACM Trans. Program. Lang. Syst.* 47, 3, Article 13 (September 2025), 35 pages.
https://doi.org/10.1145/3737282

1 Introduction

With the promise of providing efficient algorithmic solutions to many problems [14, 32, 36], some of which are traditionally believed to be intractable [60], quantum computing is the subject of intense investigation by various research communities within computer science, not least that

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" and from the ERC starting Grant CRETE (Certified Refinement Types) 101039196.
Authors' Contact Information: Andrea Colledan (corresponding author), University of Bologna, Bologna, Italy and INRIA Sophia Antipolis, Sophia Antipolis, France; e-mail: andrea.colledan@unibo.it; Ugo Dal Lago, University of Bologna, Bologna, Italy and INRIA Sophia Antipolis, Sophia Antipolis, France; e-mail: ugo.dallago@unibo.it; Niki Vazou, IMDEA Software, Pozuelo de Alarcón, Spain; e-mail: niki.vazou@imdea.org.

This work is licensed under Creative Commons Attribution International 4.0.

© 2025 Copyright held by the owner/author(s).
ACM 1558-4593/2025/9-ART13
https://doi.org/10.1145/3737282

of programming language theory [28, 49, 57]. 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 [1, 55, 58] turn out to be fundamentally new, in that the entire state of any given program (including its *control flow*) can possibly be in quantum superposition. Many other approaches are strongly inspired by classical languages and traditional programming paradigms [50, 54, 59, 71], and only allow superposition at the level of *data*, in the spirit of the so-called QRAM model [41].

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 thousand qubits have recently seen the light [12, 13, 45], it is not yet clear whether the so-called quantum advantage [51], namely the scenario in which quantum computing *empirically* solves problem instances which would require too much time to be dealt with classically, is a concrete possibility. This is due to phenomena like quantum decoherence [56], by which we mean the situation in which the state of a quantum system becomes useless, typically due to interactions with its environment. In other words, even though from a purely algorithmic point of view quantum computing undoubtedly represents an opportunity, from an architectural point of view there are big challenges, which become more acute as the size of the architecture and the number of operations to be performed grow.

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 a quantum architecture, 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 when dealing with most forms of classical computation, 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 [21], but also of the `Quipper` language [30, 31].

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` in a type-safe manner [52, 54]. Extensions include, among others, dynamic lifting [11, 25, 42] and dependent types [24, 26], but resource analysis is still a rather unexplored research direction in the Proto-Quipper community [63].

The goal of this work is to show that type systems enable the possibility of reasoning about the width of the circuits produced by a Proto-Quipper program. Specifically, we show how linear dependent types in the style of Gaboardi and Dal Lago [16, 18, 19, 27] can be adapted to Proto-Quipper, allowing to derive upper bounds on circuit widths that are parametric on 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 article is organized as follows. Section 2 informally introduces quantum circuits and explores the problem of estimating the width of circuits produced by `Quipper`, while also

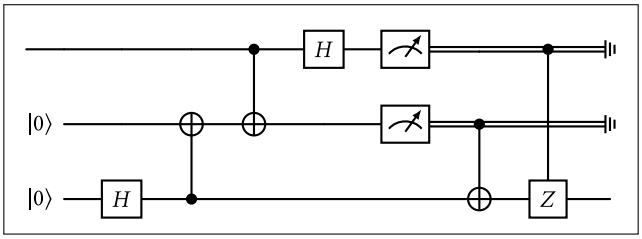

Fig. 1. An example of a quantum circuit: the quantum teleportation circuit.

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 Rios and Selinger [52], 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 [14, 46] and which is verified to do so without any ancillary qubits. Finally, Section 7 briefly describes `QuRA`, a static analysis tool based on the same linear dependent and effectful type system of the previous sections, and showcases its application to the QFT example of Section 6, as well as to a few other real-world quantum algorithms [32, 66].

To conclude this introduction, we wish to emphasize that while it is true that quantum computing can be a difficult and intimidating subject, the class of languages analyzed in this work focuses on circuit *construction*, which is an entirely classical process, paying little to no concern to the actual quantum semantics of circuit *execution*. Because of this, and due to space constraints, we refrain from providing a general introduction to quantum computing in this article. Instead, we refer the interested reader to the excellent textbooks by Nielsen and Chuang [46], Yanofsky and Mannucci [68], and Ying [69].

2 An Overview on Circuit Width Estimation

This section provides an informal overview on quantum circuits, their description in the `Quipper` language, and the problem of estimating the width of the circuits produced by a `Quipper` program.

2.1 The Quantum Circuit Model

Quantum circuits are arguably the most widely adopted model for the description of quantum computations. A quantum circuit consists of a sequence of *gates* applied to a collection of *wires*. Wires represent the individual bits and qubits involved in a computation, while gates represent elementary reversible operations applied to them. Beside gates, other *non-reversible* operations are also available, such as the initialization, discarding, or measurement of wires. Figure 1 exemplifies the notation usually employed to represent quantum circuits: Time flows from left to right, single horizontal lines represent qubit wires, double lines represent bit wires, and the various boxes and symbols that lie on wires represent different operations applied to the corresponding qubits or bits. In particular:

— The gauge symbol represents a measurement, which collapses a qubit to a classical bit.
— The other boxes represent unitary quantum gates, such as the Hadamard gate H or the Pauli Z gate.
— The ground symbol represents that a wire is discarded.

— Gates that are connected via a vertical line to a solid circle lying on a second wire are called *controlled gates* and represent the application of an operation to a wire (the *target*) conditionally on the state of the second wire (the *control*). A particularly important controlled gate is the *controlled not* (or *CNOT*) gate, which negates the state of the target qubit if the control qubit is on. Traditionally, this wire has its own notation, where the target is marked by the "⊕" symbol.

A wire that starts with $|\phi\rangle$ is initialized as part of the computation to the known state ϕ, while all other wires are input to the circuit. All wires that are not discarded as part of the computation are outputs of the circuit.

As mentioned at the end of the introduction, the act of *building* a quantum circuit is purely classical and transcends the quantum aspects of the corresponding computation, which is why we will not go into further detail about the semantics of the individual gates and operations we just introduced. However, there is one property of quantum systems, namely the *no-cloning property* of quantum states, which has a significant impact on circuit building. This property asserts that it is impossible to duplicate an arbitrary unknown quantum state, and because the wires of a quantum circuit represent, in general, quantum states, it follows that we are not allowed to duplicate or fork wires in a quantum circuit, something which is quite common in classical circuits.

We are interested in the resource consumption of quantum computations, which in the quantum circuit model corresponds to the *size* of a circuit. There are many ways to characterize the size of a circuit. A fairly immediate metric is the *width*, which is defined as the maximum number of wires active at any point in the circuit, and which corresponds to a notion of space complexity, or memory requirements of the computation, as it tells us how many quantum bits are required to run the circuit. Among the metrics that characterize running time requirements, instead, we have the *gate count*, which represents the number of gates required to define a circuit, or equivalently the number of operations that the computation needs to perform. We should also mention the more refined *depth* metric, which is the maximum number of gates encountered on any path from input to output and corresponds to *parallel* running time. As an example, the circuit in Figure 1 has a gate count of 8, a width of 3, and a depth of 6.

Quantum circuits can of course be described in a *gate-by-gate* fashion, which is what we did graphically in Figure 1 and what can be done programmatically in many low-level quantum programming languages (such as `QASM` [15]), but this approach is unwieldy and error-prone, especially when bigger circuits are involved. Furthermore, individual circuits do not usually implement algorithms. Rather, algorithms are represented by *circuit families*, which contain a circuit for each input size of the algorithm. For these reasons, it is usually preferable to rely on a *modular* approach to circuit description, where already built circuits can be manipulated and combined together into larger circuits. This approach is nowadays adopted by many high-level quantum circuits description languages and libraries, such as `Qiskit`, `Cirq`, and `Quipper`.

2.2 Width Analysis in `Quipper`

`Quipper` allows programmers to describe quantum circuits in a high-level and elegant way, using both gate-by-gate and modular 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 relying on additional temporary qubits (called *ancillae*) for the sake of reducing the circuit depth, or by recycling qubits that are no longer needed.

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

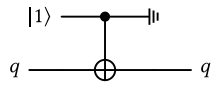

Fig. 2. A contrived Quipper implementation of the quantum not operation using an ancilla.

```
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
```

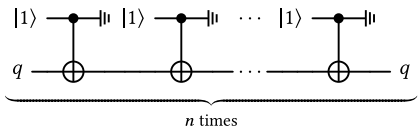

Fig. 3. A higher-order function which iterates a circuit-building function f on an input qubit q and the result of its application to the dumbNot function from Figure 2.

One feature that Quipper lacks is a methodology for *statically* and *parametrically* proving that important parameters—such as the width—of the produced circuits are below certain limits. What do we mean by all this? Suppose that a Quipper programmer writes a function, let us call it shor, which, given an integer n as input, produces as output a quantum circuit C_n, of polynomial size in $|n|$, designed to factorize n, following [60]. It is clear that this function does not correspond to a *single* circuit, because on bigger input integers shor produces circuits of increasing size. The dynamic analysis of the circuit C_n, which would simply consist in analyzing the size of C_n and which is trivial when viewed as a verification problem, can be performed only when the value of n is known. More interesting is the *static* analysis of shor, which can be performed *once and for all* and which consists in inferring a (hopefully) polynomial bound on the width of C_n that is *parametric* in n. This allows us to derive upper bounds on the number of qubits needed to run *any* instance of the program, and ultimately to understand in advance how big of an instance of the underlying problem can be *possibly* solved given a fixed amount of qubits.

In order to better 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 2. 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 a temporary qubit a, which is initialized and discarded within the body of the function. This ancillary qubit corresponds to the uppermost wire in the circuit and although it does not appear in the interface of the circuit, it clearly adds to its overall width, which is 2.

Consider now the higher-order function in Figure 3. 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.

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

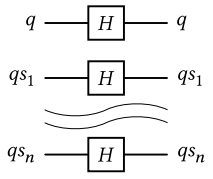

Fig. 4. The hadamardN function implements a circuit family whose circuits have width linear in the size of their input.

Is it possible to statically analyze the width of the circuit produced in output by `iter dumbNot` 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 an already non-trivial problem. The approach we propose in this article is based on two ingredients. The first is the so-called effect typing [47]. 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 such a number. 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 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 4 shows a Quipper function that returns a circuit on n qubits in which the Hadamard gate is applied to each qubit, a common preprocessing step in many quantum algorithms. It is obvious that this function works on inputs of arbitrary size, and therefore we can interpret it as a circuit family, parametrized by the length of the input list of qubits. This quantity, although certainly a natural number, is unknown statically and corresponds precisely to the width of the produced circuit. It is thus natural to wonder whether the kind of effect typing we briefly hinted at in the previous 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 by Fu et al. [26], but a simpler approach in the style of Dal Lago and Gaboardi's linear dependent types [16, 18, 19, 27] turns out to be enough for our purposes. This is precisely the route that we follow in this article. 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 $\mathsf{List}^i\ \mathsf{Qubit} \to_{i,0} \mathsf{List}^i\ \mathsf{Qubit}$, 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. Indices are better explained in Section 4, but in general we

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. 5. The Proto-Quipper types.

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	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\ M \mid (\bar{\ell}, C, \bar{k}) \mid \langle V, W\rangle$ $\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}\ V\ W$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle$

Fig. 6. The Proto-Quipper syntax.

can say that they consist of 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.

The types of Proto-Quipper are given in Figure 5. Speaking at a high level, we can say that Proto-Quipper employs a linear-nonlinear typing discipline. In particular, $w \in \{\mathsf{Bit}, \mathsf{Qubit}\}$ is a *wire type* and is linear and $\multimap$ 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.

The syntax of Proto-Quipper is given in Figure 6. Fundamentally, we are dealing with an effectful lambda calculus with bespoke constructs for manipulating circuits. A return expression turns a value into a trivial computation, while a let expression is used to sequence computations. Now, let us informally dissect the domain-specific aspects of this language, starting with the language of values. The constructs of greatest 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 [46], 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 $(\bar{\ell}, C, \bar{k})$ represents a circuit object C as a datum within the language, together with its input and output interfaces $\bar{\ell}$ and $\bar{k}$. Such a value is given parameter 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 $(\bar{\ell}, C, \bar{k})$ and appends C to the underlying circuit $\mathcal{D}$. How does apply know *where* exactly in $\mathcal{D}$ to apply 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 C is supposed to be attached.

The language is expected to be endowed with constant boxed circuits corresponding to fundamental gates and operations (e.g., Hadamard, CNOT, and initialization), but the programmer

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

Fig. 7. An example Proto-Quipper program. INIT_1, CNOT, and DISCARD are primitive boxed circuits implementing the corresponding elementary operations.

$$
\begin{aligned}
\mathit{rev} \triangleq\ & \mathsf{let}\ \mathit{revStep} = \mathsf{lift}\ \lambda\langle \mathit{revList}, q\rangle_{\mathsf{List\ Qubit}\otimes\mathsf{Qubit}}.\mathsf{return}\ (\mathsf{cons}\ q\ \mathit{revList}) \\
& \mathsf{in}\ \mathsf{fold}\ \mathit{revStep}\ \mathsf{nil}
\end{aligned}
$$

Fig. 8. An example of the use of fold: the function that reverses a list.

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 C, which is then returned by the box operator as a boxed circuit $(\bar{\ell}, C, \bar{k})$.

Figure 7 shows the Proto-Quipper term corresponding to the `Quipper` program in Figure 2, as an example of the use of the language. Note that let $\langle x, y\rangle = M$ in N is syntactic sugar for let $z = M$ in let $\langle x, y\rangle = z$ in N. The *dumbNot* function is given type Qubit $\multimap$ Qubit and builds the circuit shown in Figure 2 when applied to an argument.

On the classical side of things, it is worth mentioning that Proto-Quipper as presented in this section does *not* support general recursion. A form of structural recursion on lists is instead provided via a built-in fold constructor, which takes as argument a (duplicable) step function of type $!((B \otimes A) \multimap B)$, an initial accumulator of type B, and constructs a function of type List $A \multimap B$. Although this is not enough to recover the full power of general recursion, it appears that fold is enough to describe many quantum algorithms. Figure 8 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 $\lambda z_{A\otimes B}.$let $\langle x, y\rangle = z$ in M.

To conclude this section, we just remark that all of the `Quipper` programs shown in Section 2 can be encoded in Proto-Quipper. However, Proto-Quipper's system of simple types is 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 runtime, but this approach amounts to *testing*, not *verifying* the program, and it tells us absolutely nothing about how the size of the circuit depends on the size of the input.

4 Bringing Together 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

The types and syntax of Proto-Quipper-R are given in Figure 9. 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 [22, 53]. In our case, we are primarily concerned with

Types	*TYPE*	A, B	$::= \mathbb{1} \mid w \mid !A \mid A \otimes B \mid A \multimap_{I,J} B \mid \mathsf{List}^I A \mid \mathsf{Circ}^I(T,U)$
Param. types	*PTYPE*	P, R	$::= \mathbb{1} \mid !A \mid P \otimes R \mid \mathsf{List}^I P \mid \mathsf{Circ}^I(T,U)$
Bundle types	*BTYPE*	T, U	$::= \mathbb{1} \mid w \mid T \otimes U \mid \mathsf{List}^I T$
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	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\bar{\ell}, C, \bar{k}) \mid \langle V, W\rangle$ $\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}_i\ V\ W$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{cons}\ \bar{\ell}\ \bar{k}$
Indices	*INDEX*	I, J	$::= i \mid n \mid I + J \mid I - J \mid I \cdot J \mid \max(I, J) \mid \max_{i<I} J$

Fig. 9. Proto-Quipper-R syntax and types.

Interpretation of well-formed indices $\boxed{[\![\Theta \vdash I]\!] : \mathbb{N}^{|\Theta|} \to \mathbb{N}}$

$$
\begin{aligned}
[\![\Theta \vdash n]\!](n_1, \ldots, n_{|\Theta|}) &= n, \\
[\![\Theta, i \vdash i]\!](n_1, \ldots, n_{|\Theta|}, n_i) &= n_i, \\
[\![\Theta \vdash I + J]\!](n_1, \ldots, n_{|\Theta|}) &= [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|}) + [\![\Theta \vdash J]\!](n_1, \ldots, n_{|\Theta|}), \\
[\![\Theta \vdash I \cdot J]\!](n_1, \ldots, n_{|\Theta|}) &= [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|})[\![\Theta \vdash J]\!](n_1, \ldots, n_{|\Theta|}), \\
[\![\Theta \vdash I - J]\!](n_1, \ldots, n_{|\Theta|}) &= \max(0, [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|}) - [\![\Theta \vdash J]\!](n_1, \ldots, n_{|\Theta|})), \\
[\![\Theta \vdash \max(I, J)]\!](n_1, \ldots, n_{|\Theta|}) &= \max([\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|}), [\![\Theta \vdash J]\!](n_1, \ldots, n_{|\Theta|})), \\
[\![\Theta \vdash \max_{i<I} J]\!](n_1, \ldots, n_{|\Theta|}) &= \max\{[\![\Theta, i \vdash J]\!](n_1, \ldots, n_{|\Theta|}, n) \mid 0 \le n < [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|})\}.
\end{aligned}
$$

Fig. 10. Interpretation of well-formed indices.

circuit width, which means that the natural starting point of our extension of Proto-Quipper is precisely the circuit type: The elements of $\mathsf{Circ}^I(T,U)$ are all 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 number. 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 $I \le J$, and lastly $\max_{i<I} J$ is the maximum for i going from 0 (included) to I (excluded) of J, where i can occur in J. Note that $I = 0$ implies $\max_{i<I} J = 0$.

Let Θ be a set of index variable names, which we call an *index context*. We say that an index I is *well-formed under context* Θ, and we write $\Theta \vdash I$, when all of its free variables occur in Θ. Figure 10 provides a more formal interpretation of well-formed index terms as functions from $\mathbb{N}^{|\Theta|}$ to $\mathbb{N}$.

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 might seem quite restrictive in a generic scenario, it allows us to include lists of labels among wire bundles, something that was not possible with simple lists. This is due to the fact that sized lists are effectively isomorphic to finite tensors, and therefore a sized list of labels represent a wire bundle of known size, whereas the same is not true for a simple list of labels. 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 wires 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.

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*	$C, \mathcal{D}$	$::= id_Q \mid C; g(\bar{\ell}) \to \bar{k}$

Fig. 11. CRL syntax and types.

This variable appears locally in the type of the step function, so as to allow each fold iteration 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 enable reasoning 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 [52], but this approach makes it particularly hard to reason about their intensional properties, such as their 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.

Figure 11 introduces the **Circuit Representation Language (CRL)** which we use as the target of circuit building in Proto-Quipper-R. Wire bundles are exactly as in Figure 9 and represent arbitrary structures of wires, while circuits themselves are defined very simply as sequences 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 means 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 $C; g(\bar{\ell}) \to \bar{k}$ represents the application of the elementary operation g to the wires identified by $\bar{\ell}$ among the outputs of C. Operation g outputs the wire bundle $\bar{k}$, whose labels become part of the outputs of the overall circuit. 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, which is defined in the natural way.

Definition 4.1 (Circuit Concatenation). We define the *concatenation of* C *and* $\mathcal{D}$, or $C :: \mathcal{D}$, as

$$C :: id_Q = C, \tag{1}$$

$$C :: (\mathcal{D}; g(\bar{\ell}) \to \bar{k}) = (C :: \mathcal{D}); g(\bar{\ell}) \to \bar{k}. \tag{2}$$

4.2.1 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

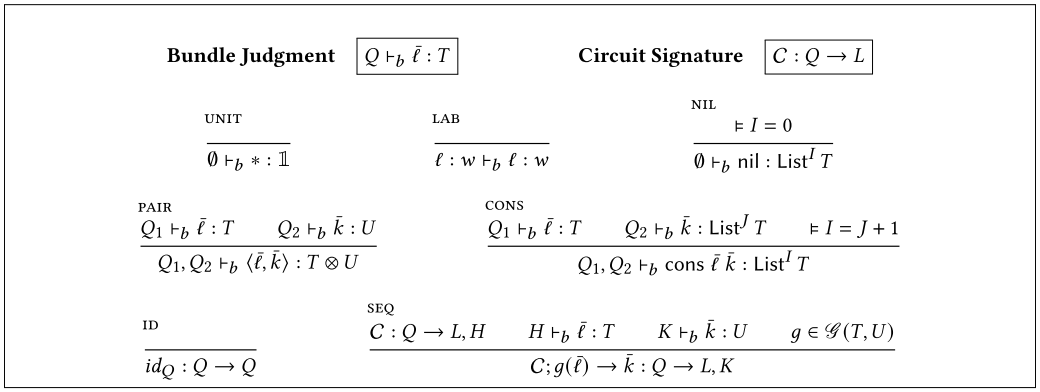

Fig. 12. The CRL type system.

circuits which allows us to derive judgments of the form $C : Q \to L$, which informally read "circuit C is well-typed with input label context Q and output label context L."

The typing rules for CRL are given in Figure 12. We call $Q \vdash_b \bar{\ell} : T$ a *bundle 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 $\mathsf{List}^{0.5}\, T$. 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 application of an elementary operation g is well-typed whenever g only acts on labels occurring in the outputs of C (those in $\bar{\ell}$, that is in H), produces in output labels that do not clash with the remaining outputs of 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 $\mathscr{G}(\mathsf{Qubit}, \mathsf{Qubit})$, while the measurement of a qubit is in $\mathscr{G}(\mathsf{Qubit}, \mathsf{Bit})$.

4.2.2 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.

Definition 4.2 (Circuit Width). We define the *width* of a CRL circuit C, written $\mathrm{width}(C)$, as follows:

$$\mathrm{width}(id_Q) = |Q|, \tag{3}$$

$$\mathrm{width}(C; g(\bar{\ell}) \to \bar{k}) = \mathrm{width}(C) + \max(0, \mathrm{new}(g) - \mathrm{discarded}(C)), \tag{4}$$

$$\mathrm{discarded}(C) = \mathrm{width}(C) - \mathrm{outputs}(C), \tag{5}$$

where $|Q|$ is the number of labels in Q, $\mathrm{new}(g)$ represents the net number of new wires initialized by g, and $\mathrm{outputs}(C)$ represents the number of outputs of C. Note that one expects $\mathrm{new}(g)$ to be equal to the difference between the number of labels in $\bar{k}$ and those in $\bar{\ell}$. The overarching idea behind this definition is that whenever we require new wires in our computation, we first try to reuse as many previously discarded wires as possible. As long as we can do this ($\mathrm{new}(g) \leq \mathrm{discarded}(C)$),

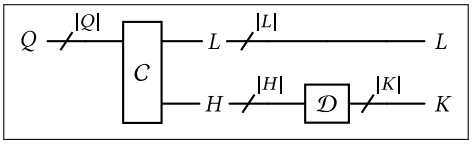

Fig. 13. The kind of scenario described by Proposition 4.3.

the initializations do not add to the total width of the circuit. Otherwise ($\text{new}(g) > \text{discarded}(C)$) we must actually create new wires, increasing the overall width of the circuit.

Another characterization of width can be obtained by expanding the definition of $\text{discarded}(C)$:

$$\begin{aligned} \text{width}(C; g(\bar{\ell}) \to \bar{k}) &= \text{width}(C) + \max(0, \text{new}(g) - \text{discarded}(C)) && (6)\\ &= \text{width}(C) + \max(0, \text{new}(g) - \text{width}(C) + \text{outputs}(C)) && (7)\\ &= \max(\text{width}(C), \text{new}(g) + \text{outputs}(C)). && (8) \end{aligned}$$

Let $|\bar{\ell}|$ be the number of labels in a bundle $\bar{\ell}$. Considering that $\text{new}(g) = |\bar{k}| - |\bar{\ell}|$, we have

$$\text{width}(C; g(\bar{\ell}) \to \bar{k}) = \max(\text{width}(C), |\bar{k}| + \text{outputs}(C) - |\bar{\ell}|). \tag{9}$$

Since $|\bar{k}|$ is the number of outputs of g and $\text{outputs}(C) - |\bar{\ell}|$ is the number of outputs of C that are unaffected by g, we can describe the width of $C; g(\bar{\ell}) \to \bar{k}$ as the maximum between the width of C and the number of outputs after the application of g. This characterization generalizes to the concatenation of circuits in the form of the fundamental property which is stated in Proposition 4.3 and illustrated in Figure 13. We use this result pervasively in proving the correctness of Proto-Quipper-R in Section 5.

Proposition 4.3 (Appending). *Given $C : Q \to L, H$ and $\mathcal{D} : H \to K$ such that the labels shared by C and $\mathcal{D}$ are all and only those in H, we have*

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

Proof. By induction of the derivation of $\mathcal{D} : H \to K$ and case analysis on $\mathcal{D}$. The case $\mathcal{D} \equiv id_H$ is trivial. The inductive case $\mathcal{D} \equiv \mathcal{D}'; g(\bar{\ell}) \to \bar{k}$ is still straightforward, but it requires further case splitting on whether $\text{new}(g) \leq \text{discarded}(C :: \mathcal{D})$ or $\text{new}(g) > \text{discarded}(C :: \mathcal{D})$. It also requires to prove the intuitive notion that the width of a circuit is bounded from below by both the number of its inputs and the number of its outputs. □

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* [47], revolving around a type judgment of the form:

$$\Theta; \Gamma; Q \vdash_c M : A; I, \tag{10}$$

which intuitively reads "for all assignments of natural numbers to 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. This is an *upper bound* on the width of the circuit described by M, as opposed to the *exact* width of such circuit, because we have to contemplate the possibility for M to exhibit branching behavior, and thus produce different circuits whose sizes depend non-arithmetically on the classical parameters in Θ. Although Proto-Quipper-R is fairly limited in terms of control flow, the existence of a fold operator with a default case is enough to warrant this overapproximation.

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 of the form $\Theta; \Gamma; Q \vdash_v V : A$ is introduced for values, which are effect-less.

The rules for deriving typing judgments are those in Figure 14, where Γ_1, Γ_2 denotes the union of two contexts with disjoint domains. The well-formedness judgment $\Theta \vdash I$ is lifted to types and typing contexts in the natural way, so we omit formal derivation rules. Among interesting typing rules, we can see how the CIRC rule bridges between CRL and Proto-Quipper-R. A boxed circuit $(\bar{\ell}, C, \bar{k})$ is well-typed with type $\mathsf{Circ}^I(T, U)$ when C is no wider than the quantity denoted by I, $C : Q \to L$ and $\bar{\ell}, \bar{k}$ contain all and only the labels in Q and L, respectively, acting as a language-level interface to 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 its argument be duplicable and 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.

4.3.1 Wire Count. Many of the rules in Figure 14 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 correspond to wires in the underlying circuit, and these wires have an intrinsic 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.

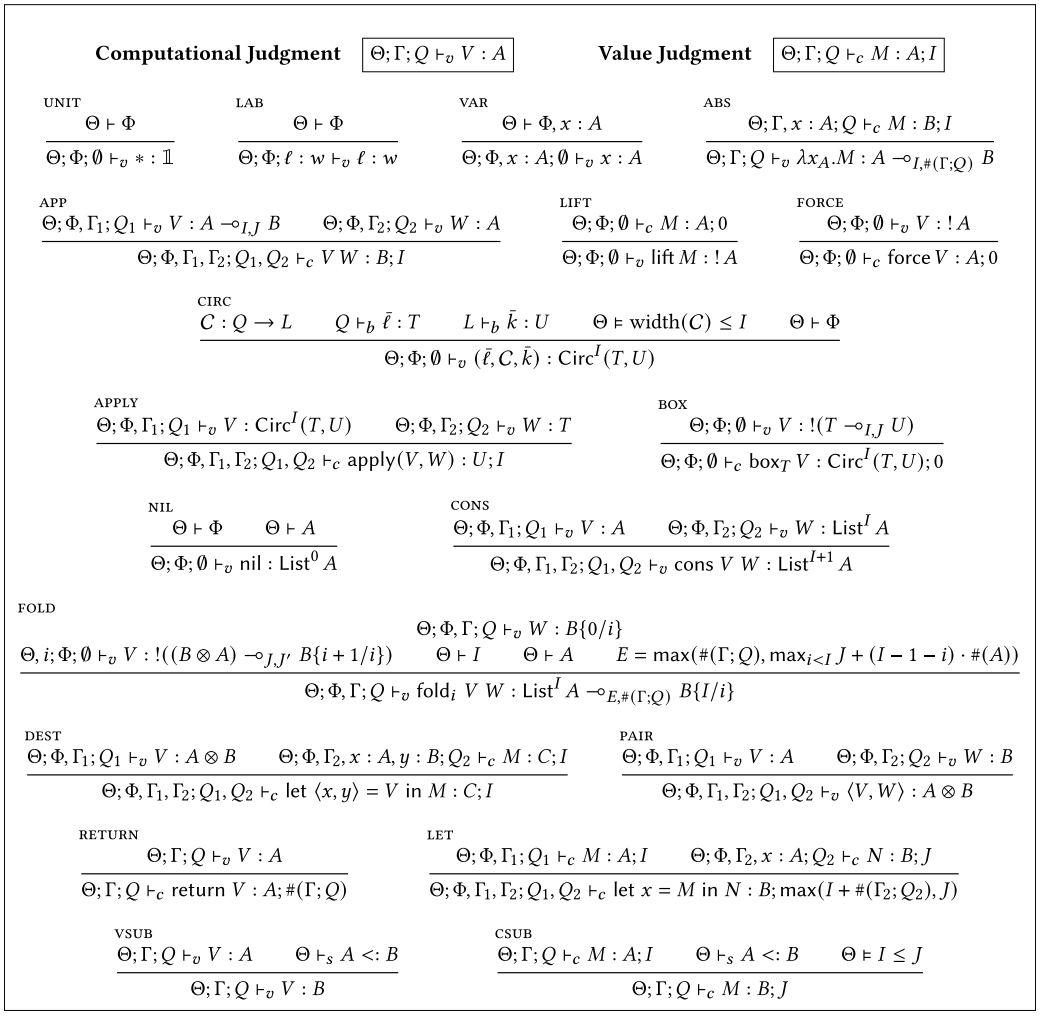

Fig. 14. Proto-Quipper-R type system.

Definition 4.4 (Wire Count). We define the *wire count of type A*, or $\#(A)$, as an index such that

$$\#(\mathbb{1}) = \#(!\,A) = \#(\mathsf{Circ}^I(T, U)) = 0, \tag{11}$$

$$\#(w) = 1, \tag{12}$$

$$\#(A \otimes B) = \#(A) + \#(B), \tag{13}$$

$$\#(A \multimap_{I,J} B) = J, \tag{14}$$

$$\#(\mathsf{List}^I\ A) = I \cdot \#(A). \tag{15}$$

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.

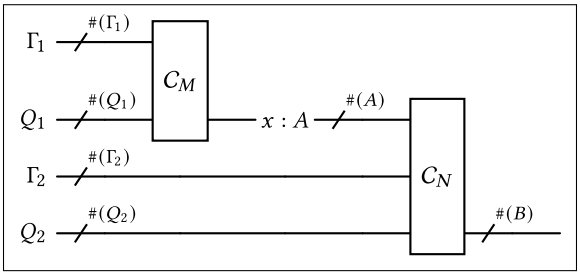

Fig. 15. The shape of a circuit built by a let expression.

The LET rule is another rule in which wire counts are essential. The two terms M and N in let $x = M$ in N build the circuits C_M and 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 expression has width bounded by $\max(I, J)$, but this fails to take into account the fact that while M is building 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 C_N. These wires must flow alongside 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 $\max(I + \#(\Gamma_2; Q_2), J)$. This situation is better illustrated in Figure 15.

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 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 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 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 $\max_{i<I} J + (I - 1 - i) \cdot \#(A)$. Once again, a graphical representation of this scenario is given in Figure 16.

4.3.2 Subtyping. Notice that Proto-Quipper-R's type system includes two subsumption rules, 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 assignments of natural numbers to the index variables in Θ, A is a subtype of B." We derive this kind of judgments by the rules in Figure 17.

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, defined as follows.

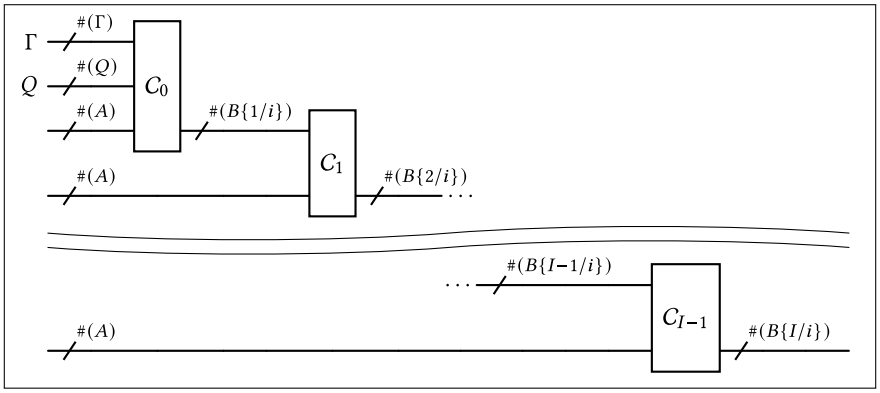

Fig. 16. The shape of a circuit built by a fold applied to an input list of type $\mathsf{List}^I\, A$.

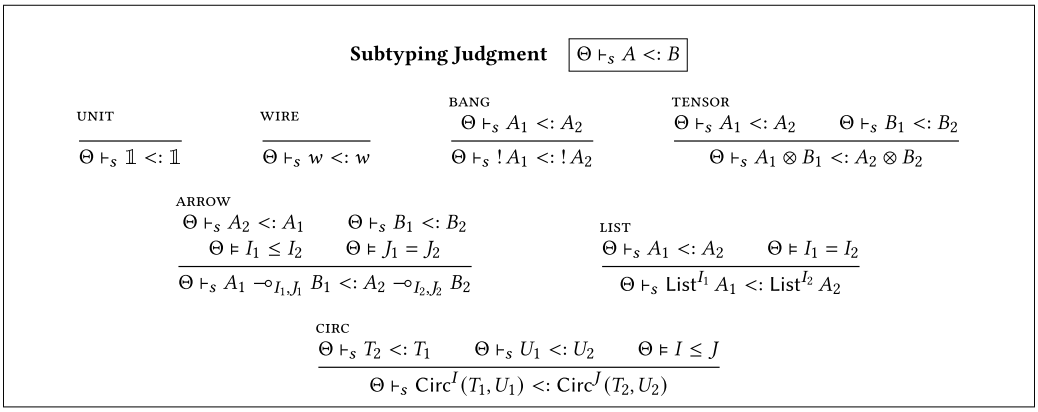

Fig. 17. Proto-Quipper-R subtyping rules.

Definition 4.5 (Semantic Judgment). We say that I *is lesser or equal than* J *for all assignment of the index variables in* Θ, and we write $\Theta \vDash I \le J$, when $\Theta \vdash I, \Theta \vdash J$ and for all $n_1, \ldots, n_{|\Theta|} \in \mathbb{N}$,

$$[[\Theta \vdash I]](n_1, \ldots, n_{|\Theta|}) \le [[\Theta \vdash J]](n_1, \ldots, n_{|\Theta|}),$$

assuming $\Theta = i_1, \ldots, i_{|\Theta|}$.

Consequently, $\Theta \vDash I = J$ is shorthand for "$\Theta \vDash I \le J$ and $\Theta \vDash J \le I$" and $\vDash I = J$ is shorthand for $\emptyset \vDash I = J$. We purposefully leave the decision procedure of this kind of judgments unspecified. In Section 7, we discuss how they can be effectively discharged to an SMT solver [10].

4.4 Operational Semantics

Operationally speaking, it does not make sense, in any Proto-Quipper dialect, 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, $(C, M) \Downarrow (\mathcal{D}, V)$ means that M evaluates to V and turns C into $\mathcal{D}$ as a side effect.

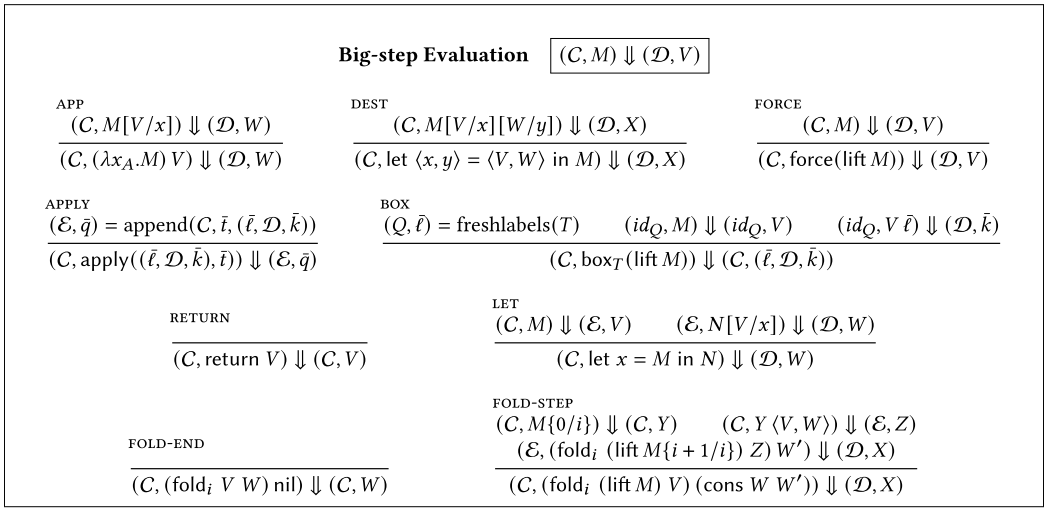

Fig. 18. Proto-Quipper-R big-step operational semantics.

The rules for evaluating configurations are given in Figure 18, where C, $\mathcal{D}$, and $\mathcal{E}$ are circuits, M and N are terms, while V, W, X, Y, and Z are values. Note that, operationally speaking, let is associative and return acts as its identity. Most 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 C and the applicand $\mathcal{D}$ is delegated entirely to the append function, which is given in Definition 4.7. Before we examine the append function, however, consider that when we deal with circuit objects we are not really interested in the concrete labels that occur in them, but rather in the *structure* that they convey. For this reason, we introduce the following notion of *equivalent circuits*.

Definition 4.6 (Circuit Equivalence). We say that two boxed circuits $(\bar{\ell}, C, \bar{k})$ and $(\bar{t}, \mathcal{D}, \bar{q})$ are *equivalent*, and we write $(\bar{\ell}, C, \bar{k}) \cong (\bar{t}, \mathcal{D}, \bar{q})$, when there exists a renaming ρ of labels such that $\rho(\bar{\ell}) = \bar{t}$, $\rho(\bar{k}) = \bar{q}$, and $\rho(C) = \mathcal{D}$.

We can now move on to the definition of append, where the notion of circuit equivalence is used to instantiate the generic input interface of a boxed circuit with the actual labels that it is going to be appended to, and to ensure that there are no name clashes between the appended circuit and the underlying circuit.

*Definition 4.7 (*append*).* We define *the append of* $(\bar{\ell}, \mathcal{D}, \bar{k})$ *to* C *on* $\bar{t}$, written $\text{append}(C, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))$, as a pair of circuit object and wire bundle computed as follows:

(1) Find $(\bar{t}, \mathcal{D}', \bar{q}) \cong (\bar{\ell}, \mathcal{D}, \bar{k})$ such that the labels shared by C and $\mathcal{D}'$ are exactly those in $\bar{t}$,
(2) Let $\mathcal{E} = C :: \mathcal{D}'$,
(3) Return $(\mathcal{E}, \bar{q})$.

On the other hand, the semantics of a term of the form $\text{box}_T(\text{lift } M)$ relies on the freshlabels function. What freshlabels does is take as input a bundle type T and instantiate fresh $Q, \bar{\ell}$ such that $Q \vdash_b \bar{\ell} : T$. 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 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 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 5.1 (Well-Typed Configuration). We say that configuration (C, M) is *well-typed with input Q, type A, width I, and output L*, and we write $Q \vdash (C, M) : A; I; L$, whenever $C : Q \to L, H$ for some H such that $\emptyset; \emptyset; H \vdash_c M : A; I$. We write $Q \vdash (C, V) : A; L$ whenever $C : Q \to L, H$ 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 $Q \vdash (C, M) : A; I; L$ evaluates to some configuration $(\mathcal{D}, V)$, that $Q \vdash (\mathcal{D}, V) : A; L$ and that $\mathcal{D}$ is obtained from 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, before we move onto these proofs, let us show that the fundamental intuition behind our formal system holds. In other words, let us prove that all of the judgments so far presented that depend on an index context Θ remain valid for any assignment of natural numbers to the variables in Θ. We actually prove the slightly more general result that such judgments remain valid under any substitution of well-formed indices for index variables. Let $\Pi \triangleright R$ denote that judgment R is the conclusion of a derivation Π.

LEMMA 5.2 (WELL-FORMEDNESS REFINEMENT). *Let Π be a well-formedness derivation and let I be an index such that $\Theta \vdash I$. We have that*

$$\Pi \triangleright \Theta, i \vdash J \implies \Theta \vdash J\{I/i\},$$
$$\Pi \triangleright \Theta, i \vdash A \implies \Theta \vdash A\{I/i\}.$$

PROOF. By induction on the size of Π and case analysis on the rule used to derive its conclusion. □

LEMMA 5.3 (SEMANTIC REFINEMENT). *Let I be an index such that $\Theta \vdash I$. We have that*

$$\Theta, i \vDash J \le E \implies \Theta \vDash J\{I/i\} \le E\{I/i\}. \tag{16}$$

PROOF. Using Lemma 5.2, we show that for all $\Theta, i \vdash J$ and $\Theta \vdash I$, we have $\Theta \vdash J\{I/i\}$ and $[\![\Theta \vdash J\{I/i\}]\!](n_1, \ldots, n_{|\Theta|}) = [\![\Theta, i \vdash J]\!](n_1, \ldots, n_{|\Theta|}, [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|}))$. By Definition 4.5, $\Theta, i \vDash J \le E$ entails $\Theta, i \vdash J$ and $\Theta, i \vdash E$, as well as $[\![\Theta, i \vdash J]\!](n_1, \ldots, n_{|\Theta|}, n) \le [\![\Theta, i \vdash E]\!](n_1, \ldots, n_{|\Theta|}, n)$, for all $n_1, \ldots, n_{|\Theta|}, n \in \mathbb{N}$, assuming $\Theta = i_1, \ldots, i_{|\Theta|}$. By choosing $n = [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|})$,

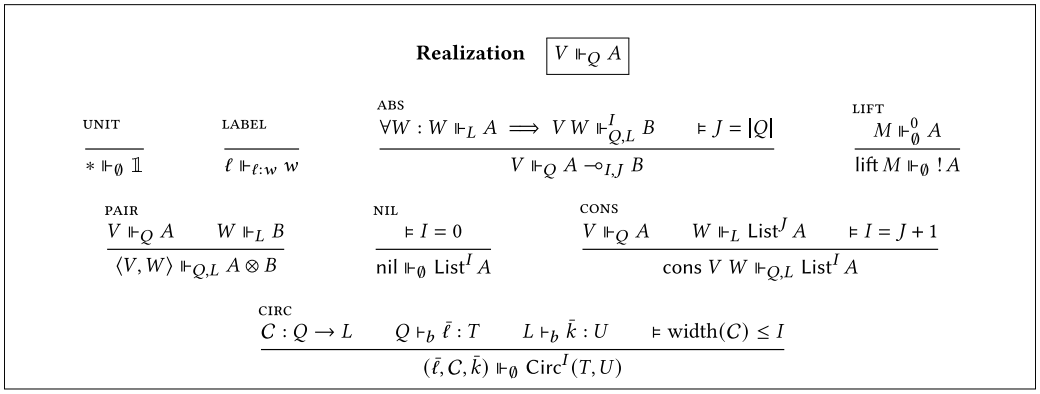

Fig. 19. Realization rules.

we get that for all $n_1, \ldots, n_{|\Theta|} \in \mathbb{N}$, $[\![\Theta, i \vdash J]\!](n_1, \ldots, n_{|\Theta|}, [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|})) \le [\![\Theta, i \vdash E]\!](n_1, \ldots, n_{|\Theta|}, [\![\Theta \vdash I]\!](n_1, \ldots, n_{|\Theta|}))$, that is, $[\![\Theta \vdash J\{I/i\}]\!](n_1, \ldots, n_{|\Theta|}) \le [\![\Theta \vdash E\{I/i\}]\!](n_1, \ldots, n_{|\Theta|})$, which concludes the claim. □

Lemma 5.4 (Subtyping Refinement). *Let I be an index such that $\Theta \vdash I$. We have that*

$$\Theta, i \vdash_s A <: B \implies \Theta \vdash_s A\{I/i\} <: B\{I/i\}. \tag{17}$$

Proof. By induction on the derivation of $\Theta, i \vdash_s A <: B$ and case analysis on the rule used to derive its conclusion, making use of Lemmata 5.2 and 5.3. □

Lemma 5.5 (Typing Refinement). *Let Π be a typing 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 and case analysis on the rule used to derive its conclusion, using Lemmata 5.2–5.4. □

Given Lemmata 5.2 through 5.5, if we take I to be an arbitrary number $n \in \mathbb{N}$, we obtain precisely our fundamental intuition. We can now move onto subject reduction and total correctness. Both these results rely on a central lemma and on the mutual notions of *realization* and *reducibility*, which we first give formally. We define $V \Vdash_Q A$, which reads V *realizes* A *under* Q, as the smallest relation closed under the rules in Figure 19, while reducibility is defined as follows.

Definition 5.6 (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 C such that $C : L \to Q, H$, there exist $\mathcal{D}, V$ such that

(1) $(C, M) \Downarrow (C :: \mathcal{D}, V)$,
(2) $\vDash \mathsf{width}(\mathcal{D}) \le I$,
(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 [61]. The intuition is pretty straightforward: A term is reducible with width I if it evaluates correctly when paired with any circuit C which provides its free labels and if it extends 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 of both 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 5.7 (Fundamental Lemma). *Let* Π *be a type derivation. For all* $\theta \vDash \Theta$ *and* $\gamma \vDash_Q \theta(\Gamma)$*, we have that*

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

Proof. By induction on the size of Π and case analysis on the rule used to derive its conclusion. Most cases are straightforward once we prove that for all V, Q, and A, $V \Vdash_Q A$ implies $\#(A) = |Q|$ and that whenever a substitution γ implements Γ_1, Γ_2, then $\gamma = \gamma_1 \circ \gamma_2$ for some γ_1, γ_2 that implement Γ_1 and Γ_2, respectively. On the other hand, nontrivial proof cases include

- The let case, which requires Proposition 4.3 and the associativity of :: to deal with the sub-circuits built by the two sub-terms of the let expression.
- The apply case, which requires showing that wire bundles of the same type can always be renamed into each other and that renaming labels inside a boxed circuit preserves realization.
- The vsub and csub cases, which require showing that for all V, Q, A, and B, $V \Vdash_Q A$ and $\vdash_s A <: B$ entail $V \Vdash_Q B$.
- The fold case, which requires proving that for all X realizing the argument type of $\mathsf{fold}_i\ V\ W$, let it be $\mathsf{List}^J B$, we have that $(\mathsf{fold}_i\ X\ W)\ X$ is reducible. This requires further induction on $X \Vdash_H \mathsf{List}^J B$ and the use of Proposition 4.3.

□

Lemma 5.7 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 5.7, we can proceed to proving the aforementioned results of subject reduction and total correctness. We start with the former, which unsurprisingly requires a substitution lemma.

Lemma 5.8 (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 and case analysis on the rule used to derive its conclusion. □

Theorem 5.9 (Subject Reduction). *If* $Q \vdash (C, M) : A; I; L$ *and* $(C, M) \Downarrow (\mathcal{D}, V)$, *then* $Q \vdash (\mathcal{D}, V) : A; L$.

Proof. By induction on the derivation of $(C, M) \Downarrow (\mathcal{D}, V)$ and case analysis on the rule used to derive its conclusion. Lemma 5.8 is essential to the app, dest, and let cases, while Lemma 5.5 is essential to the fold-step case. Lemma 5.7 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 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 5.10 (Total Correctness). *If* $Q \vdash (C, M) : A; I; L$, *then there exist* $\mathcal{D}, V$ *such that* $(C, M) \Downarrow (C :: \mathcal{D}, V)$ *and* $\vDash \mathrm{width}(\mathcal{D}) \leq I$.

Proof. By definition, $Q \vdash (C, M) : A; I; L$ entails that $C : Q \to L, H$ and $\emptyset; \emptyset; H \vdash_c M : A; I$. Since an empty context is trivially implemented by an empty closing substitution, by Lemma 5.7 we get $M \Vdash^{I}_{H} A$, which by definition entails that there exist $\mathcal{D}, V$ such that $(C, M) \Downarrow (C :: \mathcal{D}, V)$ and $\vDash \mathrm{width}(\mathcal{D}) \leq I$. □

6 An Example: The QFT

This section provides an example of how realistic type derivations are carried out in Proto-Quipper-R. In particular, we use our language to implement the QFT algorithm [14, 46] and verify that the circuits it produces have width no greater than the size of their input. At an intuitive level, the QFT is the quantum analogue of the Discrete Fourier Transform [29]. It is an ubiquitous subroutine in quantum computing and it plays a fundamental role in many mainstream algorithms, such as Shor's algorithm [60], quantum phase estimation [46], quantum modular arithmetic [2], and the HHL algorithm [36].

A Proto-Quipper-R implementation of the QFT algorithm is given in Figure 20. 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 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 *rev* stand for regular language terms which implement respectively the linear list length and reverse functions. They have type $qlen : \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} (\mathsf{Nat} \otimes \mathsf{List}^i\ \mathsf{Qubit})$ and $rev : \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\ \mathsf{Qubit}$ 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 *rotate n* function, which applies controlled rotations between appropriate pairs of qubits and has type

$$(\mathsf{Qubit} \otimes \mathsf{List}^e\ \mathsf{Qubit}) \otimes \mathsf{Qubit} \multimap_{e+2,0} \mathsf{Qubit} \otimes \mathsf{List}^{e+1}\ \mathsf{Qubit}, \tag{18}$$

meaning that *rotate n* rearranges the structure of its inputs, but overall does not introduce any new wire. We fold this function starting from an accumulator $\langle q, \mathsf{nil} \rangle$ of type $\mathsf{Qubit} \otimes \mathsf{List}^0\ \mathsf{Qubit}$,

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

Fig. 20. A Proto-Quipper-R implementation of the QFT circuit family. The usual syntactic sugar is employed.

meaning that we can give $\mathsf{fold}_j\ (\mathsf{lift}(rotate\ n))\ \langle q, \mathsf{nil}\rangle$ type as follows:

$$
\frac{\begin{array}{c}\textsc{fold} \\ i, j, e; n : \mathsf{Nat}; \emptyset \vdash_v \mathsf{lift}(rotate\ n) : !((\mathsf{Qubit} \otimes \mathsf{List}^e\ \mathsf{Qubit}) \otimes \mathsf{Qubit} \multimap_{e+2,0} \mathsf{Qubit} \otimes \mathsf{List}^{e+1}\ \mathsf{Qubit}) \\ i, j; q : \mathsf{Qubit}; \emptyset \vdash_v \langle q, \mathsf{nil}\rangle : \mathsf{Qubit} \otimes \mathsf{List}^0\ \mathsf{Qubit} \qquad i, j \vdash j \qquad i, j \vdash \mathsf{Qubit}\end{array}}{i, j; n : \mathsf{Nat}, q : \mathsf{Qubit}; \emptyset \vdash_v \mathsf{fold}_e\ \mathsf{lift}(rotate\ n)\ \langle q, \mathsf{nil}\rangle : \mathsf{List}^j\ \mathsf{Qubit} \multimap_{j+1,1} \mathsf{Qubit} \otimes \mathsf{List}^j\ \mathsf{Qubit}} \tag{19}
$$

where we implicitly use the fact that $i, j \vDash \mathsf{max}(1, \mathsf{max}_{e<j}\ e + 2 + (j - 1 - e) \cdot 1) = j + 1$ 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

$$
!((\mathsf{List}^j\ \mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\ \mathsf{Qubit}), \tag{20}
$$

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

$$
\frac{\begin{array}{c}\textsc{fold} \\ i, j; \emptyset; \emptyset \vdash_v qftStep : !((\mathsf{List}^j\ \mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\ \mathsf{Qubit}) \\ i; \emptyset; \emptyset \vdash_v \mathsf{nil} : \mathsf{List}^0\ \mathsf{Qubit} \qquad i \vdash i \qquad i \vdash \mathsf{Qubit}\end{array}}{i; \emptyset; \emptyset \vdash_v \mathsf{fold}_j\ qftStep\ \mathsf{nil} : \mathsf{List}^i\ \mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\ \mathsf{Qubit}}, \tag{21}
$$

where we once again implicitly simplify the arrow type using the fact that $i \vDash \mathsf{max}(0, \mathsf{max}_{j<i}\ j + 1 + (i - 1 - j) \cdot 1) = i$. 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 21, whose width is exactly 3.

To conclude this section, note that for ease of exposition *qft* actually produces the *reversed* QFT circuit, in which the gates appear in reverse order. 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 [20].

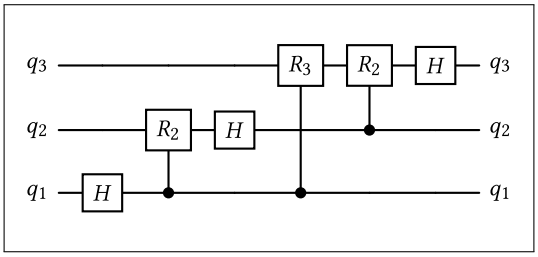

Fig. 21. The circuit of input size 3 produced by *qft* (cons q_1 cons q_2 cons q_3 nil).

7 Implementation

This section introduces the QuRA tool (IPA:/'kura/, "care"), a static analysis tool implemented based on Proto-Quipper-R's type system.[1] QuRA takes as input a program written in a concrete variant of Proto-Quipper-R's syntax and outputs the type of the program and an upper bound to the width of the circuit it builds. We limit ourselves to presenting the most relevant aspects of the implementation of QuRA and three practical applications of the tool, with no ambition to provide an exhaustive introduction to it. The interested reader is referred to QuRA's documentation, which is available alongside its source code at https://github.com/andreacolledan/qura. QuRA is implemented in `Haskell`, and at the time of writing relies on the CVC5 SMT solver [7] to discharge semantic judgments.

7.1 Internalizing Quantification over Indices

One major hurdle in implementing Proto-Quipper-R's type system as part of a tool is the necessity to internalize the meta-theoretical results on index substitution (corresponding to Lemmata 5.2 through 5.5) into the typing rules. This is because meta-theoretical index quantification, although convenient when proving soundness and correctness, is not a viable option when programming in practice: The scope of index variables is unclear and writing index-polymorphic functions is counter-intuitive and error-prone (e.g., it is really easy to incur accidental variable capture).

To internalize index quantification, we introduce explicit index abstraction and application into the language, in the form of the $\Lambda i.M$ and $V\ @\ I$ constructs, respectively. The two constructs respectively introduce and eliminate a new type form $\Pi^I_J i.A$, which very closely resembles a dependent function type in which the argument i is always an index. In the wider context of the Proto-Quipper-R type system, these two new constructs can be thought of as obeying the following typing rules:

$$\frac{\overset{IABS}{\Theta, i; \Gamma; Q \vdash_c M : A; I}}{\Theta; \Gamma; Q \vdash_v \Lambda i.M : \Pi^I_{\#(\Gamma;Q)} i.A} \qquad \frac{\overset{IAPP}{\Theta\ ; \Gamma; Q \vdash_v V : \Pi^J_E i.A} \qquad \Theta \vdash I}{\Theta; \Gamma; Q \vdash_c V@I : A\{I/i\}; J\{I/i\}}$$

This addition also influences the treatment of the fold operator: We have seen in the previous sections that fold_i binds i in the type of the step function. Now that a mechanism for explicit index binding is available, it is more convenient to separate concerns and relieve the fold operator from this responsibility. In the wake of this change, fold is also relieved from the responsibility of introducing a fresh index variable in the domain of its arrow type: Instead of taking two arguments and constructing a function, fold now takes as input three arguments V, W, and X and constructs

[1] At the time of writing, the latest version of QuRA actually implements a superset of Proto-Quipper-R's theory, with support for the analysis of metrics other than width, such as gate count and even depth. In this section, however, we will limit ourselves to an account of the width estimation capabilities of the tool.

the computation where V is folded over X, using W as the starting accumulator. This new version of fold can be thought of as obeying the following typing rule:

$$\textsc{fold}\quad \frac{\Theta;\Phi;\emptyset \vdash_v V : !(\Pi_0^0 i.(B \otimes A) \multimap_{J,0} B\{i+1/i\}) \qquad \Theta;\Phi,\Gamma_1;Q_1 \vdash_v W : B\{0/i\} \\ \Theta;\Phi,\Gamma_2;Q_2 \vdash_v X : \mathsf{List}^E\, A \qquad F = \max(\#(\Gamma_1;Q_1), \max_{i<E} J + (E-1-i)\cdot \#(A))}{\Theta;\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash_c \mathsf{fold}\ V\ W\ X : B\{E/i\};F}$$

A Proto-Quipper-R function such as $\mathsf{fold}_i\ (\mathsf{lift}\ M)\ W$ of type $\mathsf{List}^I\, A \multimap_{J,E} B$ therefore corresponds to a QuRA term of the form $\lambda x_{\mathsf{List}^I\, A}.\,\mathsf{fold}\ (\Lambda i.\,\mathsf{lift}\ M)\ W\ x$, provided that any and all free index variables in I are already available in the context.

7.2 Type Inference and SMT Solving

We implement Proto-Quipper-R's typing rules in the form of a type inference algorithm. For the sake of practicality, in the implementation of QuRA we do not distinguish syntactically between terms and values as we did in the previous sections. This has two consequences:

- The return operator is no longer needed: A standalone value simply receives a width upper bound equal to the wire count of its typing environment, as if it had been implicitly returned.
- Terms can be built from other (effectful) terms, and as such the width of the circuits built by sub-terms must be taken into consideration when computing the upper bound of a compound term, in a way similar to what we do in the LET rule in Section 4. In fact, the LET rule directly inspires the design of the new inference rules. For example, a term such as $\mathsf{apply}(M, N)$ is treated as if it were shorthand for $\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ y = N\ \mathsf{in}\ \ \mathsf{apply}(x, y)$ and the corresponding rule is designed accordingly.

That being said, QuRA performs two passes on the AST of a program, following a standard approach in the implementation of refinement type systems [39]:

(1) In the first pass we ignore indices and width upper bounds and just perform Hindley-Milner-like inference on the program. This pass is used to check for basic type errors and to annotate some of the nodes of the AST with their type, as to simplify the subsequent pass.
(2) In the second pass, we assume that the basic types are in the correct form and focus on the synthesis of index annotations and width upper bounds.

The second pass is where the actual width estimation logic takes place. As a way to exemplify more formally the implementation of the tool, an excerpt of the algorithmic inference rules for the second pass can be found in Figure 22. A judgment of the form $\Theta;\Gamma_0 \setminus \Gamma_1;Q_0 \setminus Q_1 \vdash M \Rightarrow A;I$ means that the inference algorithm takes as input Θ, Γ_0, Q_0, M and determines that M has type A and width upper bound I, consuming *some* of the linear resources contained in Γ_0 and Q_0. Indeed, alongside A and I, the algorithm also outputs Γ_1 and Q_1, which contain the linear resources in Γ_0 and Q_0, respectively, that were *not* consumed while typing M. This is a standard technique employed in the implementation of linear type systems.

7.2.1 SMT Encoding of Semantic Judgments. Notice how the semantic judgments of the form $\Theta \vDash I \leq J$ are replaced by a call to the `valid` function. This function represents a call to the SMT solver, which we delegate the proof of semantic judgments to. What $\mathtt{valid}_\Theta(I \leq J)$ does is query the validity of the logical formula $I \leq J$, where the free variables of I and J are in Θ.

This requires a nontrivial encoding of index terms in the SMTLib2 format [8]: While index variables are simply encoded as integer variables subject to non-negativity constraints and most arithmetic operations have a one-to-one correspondence with operations in the target language,

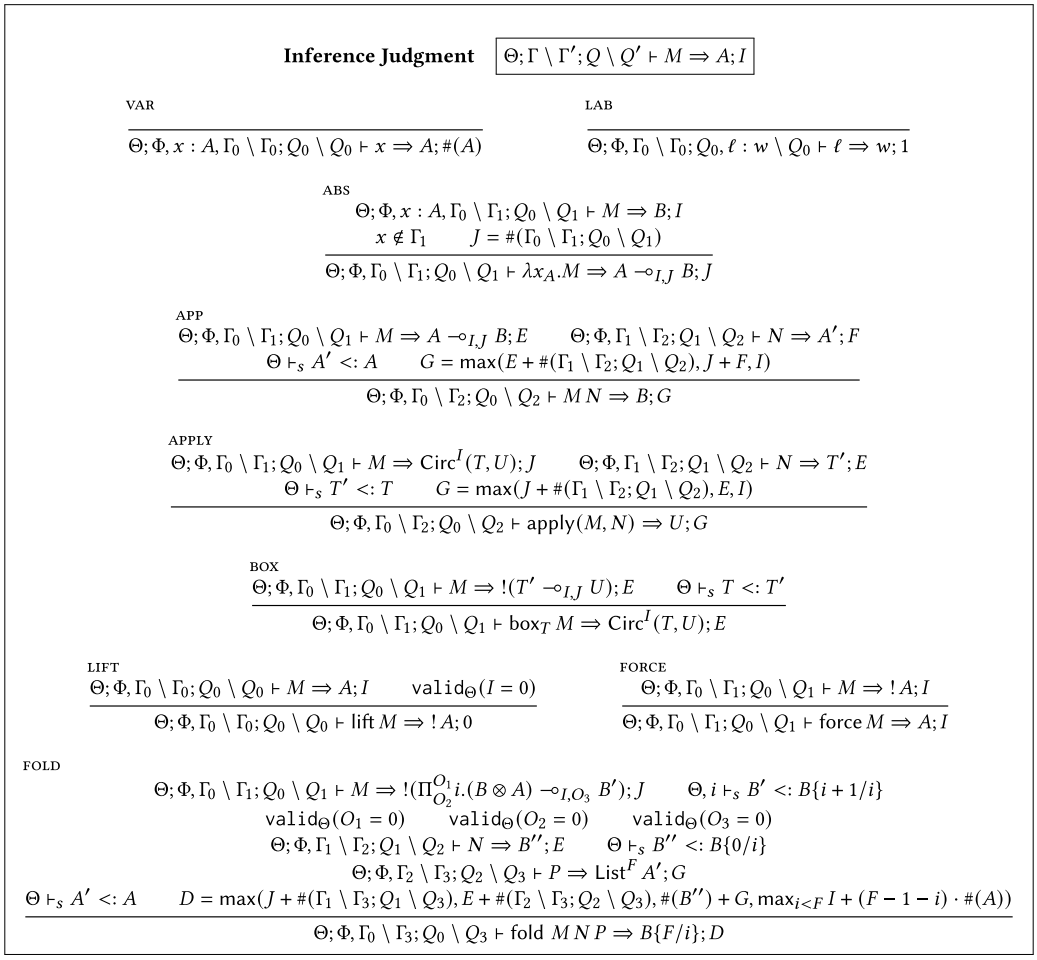

Fig. 22. An excerpt of the type inference rules.

natural subtraction, binary and bounded of maxima do not. For natural subtraction and the binary maximum, the encoding is still rather simple. Let $(\!|I|\!)_{\texttt{SMTLib}}$ denote the encoding of I as an `SMTLib2` expression. Then we have

$$(\!|I - J|\!)_{\text{SMTLib}} \triangleq (\text{ite } (< (\!|I|\!)_{\text{SMTLib}} (\!|J|\!)_{\text{SMTLib}}) \emptyset (- (\!|I|\!)_{\text{SMTLib}} - (\!|J|\!)_{\text{SMTLib}})), \tag{22}$$

$$(\!|\max(I, J)|\!)_{\text{SMTLib}} \triangleq (\text{ite } (< (\!|I|\!)_{\text{SMTLib}} (\!|J|\!)_{\text{SMTLib}}, (\!|J|\!)_{\text{SMTLib}}, (\!|I|\!)_{\text{SMTLib}}). \tag{23}$$

where `ite` is the conditional operator in `SMTLib2` (note the use of prefix notation). The case of the bounded maximum is more interesting: An expression such as $\max_{i<I} J$ cannot be encoded directly in `SMTLib2`, and a higher-order, inductive definition of the operator causes the solver to get stuck on all but the trivial cases. Rather, for each such expression we introduce two auxiliary variables `max` and `argmax`, and we constrain them to be equal to the value of $\max_{i<I} J$ and to the value of i that maximizes J, respectively. We do so through the following constraints, which we present in

standard mathematical notation for simplicity:

$$0 \leq \mathtt{max}, \tag{24}$$

$$I \leq 0 \implies \mathtt{max} = 0, \tag{25}$$

$$I > 0 \implies \mathtt{max} = J\{\mathtt{argmax}/i\}, \tag{26}$$

$$0 \leq \mathtt{argmax} < I, \tag{27}$$

$$\forall j.0 \leq j < I \implies J\{j/i\} \leq J\{\mathtt{argmax}/i\}. \tag{28}$$

With these constraints in place, every occurrence of $\mathtt{max}_{i<I}\, J$ can be replaced by the `max` variable. Lastly, we express the problem of deciding the validity of $I \leq J$ as the equivalent satisfiability problem for $\neg(I \leq J)$ and query the solver. A `sat` response denotes that $I \leq J$ is falsifiable, and thus not valid, so the corresponding judgment does not hold, while an `unsat` response proves the judgment.

7.2.2 The Limitations of SMT Solvers. Reasoning with SMT solvers can be highly efficient as long as the queried theories belong to SMT-decidable logics. Following the principles of refinement typing [39], QuRA attempts to produce queries belonging to the decidable logic of equality and linear arithmetic, while exploiting Proto-Quipper-R's typing rules and carefully designed encodings to deal with undecidable features like induction and higher-order reasoning. The aforementioned case of bounded maxima is a characteristic example of this approach: When naively implemented in SMTLib2, bounded maxima can make the solver diverge, rendering verification unpredictable. Thanks to the encoding described in Equations (24)–(28), however, queries involving bounded maxima become decidable via quantifier elimination [62].

Secondly, while SMT solvers excel in handling linear arithmetic, they can struggle with nonlinear constraints. Due to the nature of width analysis, most of the upper bounds we actually need to prove in Proto-Quipper-R are defined in terms of sums and maxima of natural numbers and index variables, which are handled gracefully by CVC5. At the same time, the SMT queries that we work with *can* contain multiplication, usually in the form of the wire count of list types, as $\#(\mathsf{List}^I\, A)$ is defined as $I \cdot \#(A)$. Even in this case, though, $\#(A)$ is almost always a constant factor, so the resulting expression remains linear. This is due to the fact that we mostly deal with lists whose elements are qubits or tuples of qubits, which have a constant wire count. In theory, if we were working with lists of lists (i.e., bidimensional qubit arrays of variable, independent dimensions), then we would incur nonlinearity, but in practice this is a very rare scenario.

Lastly, while solvers (and CVC5 in particular) are efficient, queries to them require expensive context switches. We mitigate this problem by employing a custom solver for closed constraints, which attempts to evaluate both sides of a constraint to a number according to standard arithmetic rules before checking that the queried inequality holds. Only when this simpler solver fails (usually because either side of the constraint contains an index variable) do we forward the query to the more powerful CVC5.

All things considered, we can say that the implementation of QuRA is designed to bypass the limitations of SMT solving. To some extent, this has been confirmed empirically: When using QuRA to analyze the resource requirements of real quantum algorithms (see Section 7.3) we never encountered a situation in which we could not verify a width upper bound *specifically* due to SMT solving limitations.

7.3 Practical Verification of Quantum Algorithms

We now proceed to illustrate how QuRA can be used to verify the resource consumption of three real-world quantum algorithms. Note that because QuRA fundamentally implements the theory that

Listing 1. A program implementing the QFT algorithm that can be given as input to QuRA.

```
--- Quantum Fourier Transform with width analysis

--- Relevant parameters:
--- n     : size of the input to the QFT
--- iter  : current iteration of the QFT

--- HELPER FUNCTIONS ---

-- invert the list of intermediate qubits at iteration iter
let qrev = lift forall iter.
  \reg :: List[_<iter] Qubit.
    let revStep = lift forall step.
      \(rev, q) :: (List[_<step] Qubit, Qubit).
        rev:q in
    fold(revStep, [], reg)
in

-- apply the controlled rotation gate to the target qubit trg at iteration iter
let rotate = forall iter. lift forall step.
  \((ctrls, trg), ctrl)::((List[_<step] Qubit, Qubit), Qubit).
    let (ctrl, trg) = (force cr @(iter+1-step) @0 @0) ctrl trg in
    (ctrls:ctrl, trg)
in

--- QUANTUM FOURIER TRANSFORM ---

-- apply the Quantum Fourier Transform to n qubits
let qft = lift forall n.
  \reg :: List[_<n] Qubit.
    let qftIter = lift forall iter. -- define the iteration of the QFT
      \(ctrls, trg)::(List[_<iter] Qubit, Qubit).
        let revctrls = (force qrev @iter) ctrls in
        let (ctrls, trg) = fold(rotate @iter, ([], trg), revctrls) in
        let trg = (force hadamard @0) trg in
        ctrls:trg
    in fold(qftIter, [], reg)
in

qft
```

we have presented in this article, this section is meant to showcase not only the practicality of the tool, but also the power of Proto-Quipper-R's theory.

7.3.1 The QFT Algorithm. We start with the familiar example of the QFT algorithm from Section 6, which is implemented by the program in Listing 1. The concrete syntax adopted in QuRA is reminiscent of Haskell and should be fairly self-explanatory. We just point out that "`forall i. M`" stands for the index abstraction "$\Lambda i.M$," whereas "`List[_<I]A`" stands for the sized list type "$\mathsf{List}^I\ A$." The unusual syntax for list types is due to the fact that QuRA actually supports *dependent* sized lists, which we do not use or discuss in this section. Note only that these lists grow to the right, as opposed to the left, to better support the dependent aspect. Other than that, lists are exactly as in Proto-Quipper-R.

By running QuRA on the program from Listing 1, we get the expected result that `qft` has type

$$!(\Pi_0^0 n.\mathsf{List}^n\ \mathsf{Qubit} \multimap_{n,0} \mathsf{List}^n\ \mathsf{Qubit}), \tag{29}$$

meaning that it is a duplicable function that for all n takes as input a list of n qubits, builds a circuit of width at most n, and outputs a list of n qubits, which is exactly what we derived at the end of Section 6.

7.3.2 Quantum Adder. A more interesting application of our tool is an implementation of the quantum addition algorithm due to Vedral et al. [66]. Although far from the current state of the art in terms of quantum arithmetic, this algorithm does make extensive use of ancillae, requiring $3n + 1$ qubits to add together two n-qubit quantum registers, and as such is a good testing ground for our

analysis. We omit the code (which is however available in QuRA's repository) for reasons of space, but QuRA is able to infer that the adder function, which assembles the main adder circuit, has type

$$!(\Pi_0^0 n.(\mathsf{List}^{n+1}\,\mathsf{Qubit} \otimes \mathsf{List}^{i+1}\,\mathsf{Qubit}) \multimap_{(3\cdot(n+1)+1),0} (\mathsf{List}^{n+1}\,\mathsf{Qubit} \otimes \mathsf{List}^{n+2}\,\mathsf{Qubit})), \tag{30}$$

meaning that it is a duplicable function that for any input size n takes two lists of $n+1$ qubits and builds a circuit of width at most $3(n+1)+1$ that outputs two lists of respectively $n+1$ and $n+2$ qubits (the extra qubit is needed to store the potential carry). Note that we use $n+1$ instead of n in the type of the input lists just to enforce that the input registers be non-empty.

7.3.3 Grover's Algorithm. We conclude by verifying a higher-order quantum algorithm. Grover's algorithm [32] is used to invert a function f that maps n-bit strings to Booleans, given that there is exactly one x such that $f(x) = 1$. Analyzing the resource consumption of Grover's algorithm is more challenging than the previous examples, as it does not take as input a simple register of qubits, but rather an *oracle*, that is, a subcircuit implementing the function f. In our implementation of the algorithm, which is given in Listing 2, this oracle is represented by the oracle parameter of the grover function, which has type

$$\mathsf{Circ}^{ow}(\mathsf{List}^n\,\mathsf{Qubit} \otimes \mathsf{Qubit}, \mathsf{List}^n\,\mathsf{Qubit} \otimes \mathsf{Qubit}). \tag{31}$$

This means that oracle is a boxed circuit of width at most ow acting on a register of n qubits *plus* an ancilla, which accounts for the standard way classical functions are implemented as reversible unitary operators. Note that both n and ow are parameters of the grover function, together with r, the number of iterations of the algorithm.

The algorithm itself consists of a preparation stage, followed by r *Grover iterations*, followed by a measurement stage. A Grover iteration, in turn, consists of a call to the oracle, followed by an application of the *diffusion* operator. Figure 23 shows the circuit built by the grover function when applied to an oracle of input size $n = 3$.

Let us reason briefly about the size of such a circuit. The preparation and measurement stages are uninteresting in this regard. Furthermore, because the diffusion operator has width equal to its input size, the width of the circuit built by the grover function is intuitively dominated by the width of the oracle. If we run QuRA on the program from Listing 2, we find that our intuition is confirmed, as the grover function has type

$$!(\Pi_0^0 r.\Pi_0^0 n.\Pi_0^0 ow.\,\mathsf{Circ}^{ow}(\mathsf{List}^n\,\mathsf{Qubit} \otimes \mathsf{Qubit}, \mathsf{List}^n\,\mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{\max(n+1,ow),0} \mathsf{List}^n\,\mathsf{Bit}), \tag{32}$$

which tells us that it is a duplicable function that takes as input an oracle of input size n and width ow, and builds a circuit of width at most $\max(n+1, ow)$. Note that because ow is bound to be at least $n+1$, the width of the circuit is effectively upper bounded by ow.

7.4 Benchmarks

In all of the examined example programs, few annotations besides those on function arguments are needed by the inference algorithm: Occasionally, the type of step-functions needs to be reformulated in a way that is compatible with the FOLD rule, and sometimes we need to postulate that a list is guaranteed to be non-empty. Table 1 shows the performance of QuRA on a small benchmark suite of quantum programs, which is available in the repository, under examples.

7.5 Dynamic vs. Static Analysis

In the beginning of this article we stated that Quipper is effectively used to write complex programs that translate to circuits comprising of millions of qubits. If we resort to a *dynamic* approach to

Listing 2. A program implementing Grover's algorithm.

```
--- Grover search algorithm with width analysis ---

--- Relevant parameters:
--- n     : input size of the oracle
--- ow    : width of the oracle

--- HELPER FUNCTIONS (omitted) ---
--- [...]

--- GROVER'S ALGORITHM ---

-- the diffusion operator on n qubits with ancilla a (all qubits at depth d)
let diffusion = lift forall n.
  ((\reg :: List[_<n] Qubit. \a :: Qubit.
    let reg = (force mapHadamard @n @0) reg in
    -- begin negatively controlled not
    let reg = (force mapQnot @n @0) reg in
    let (reg,a) = (force mcnot @n @0 @0) reg a in
    let reg = (force mapQnot @n @0) reg in
    -- end negatively controlled not
    let reg = (force mapHadamard @n @0) reg in
    (reg, a)))
in

-- perform a single grover iteration on n qubits, using an ancilla a and an oracle of width ow
let groverIteration = lift forall n. forall ow.
  \oracle :: Circ[ow]((List[_<n] Qubit, Qubit), (List[_<n] Qubit, Qubit)).
  \reg :: List[_<n] Qubit.
  \a :: Qubit.
    let (reg, a) = apply(oracle, (reg, a)) in
    (force diffusion @n) reg a
in

-- run Grover's algorithm on an oracle of input size n and width ow, for r iterations
let grover = lift forall r. forall n. forall ow.
  \oracle :: Circ[ow]((List[_<n] Qubit, Qubit), (List[_<n] Qubit, Qubit)).
    -- prepare working qubits
    let wqs = force qinit0Many @n in
    let wqs = (force mapHadamard @n @0) wqs in
    -- prepare ancilla
    let a = force qinit1 in
    let a = (force hadamard @0) a in
    -- iterate Grover's algorithm
    let iteration = lift forall step. \((wqs, a), _) :: ((List[_<n] Qubit, Qubit), ()).
      (force groverIteration @n @ow) oracle wqs a in
    let (wqs, a) = fold(iteration, (wqs, a), range @r) in
    let _ = (force qdiscard @0) a in
    (force mapMeasure @n @0) wqs
in

grover
```

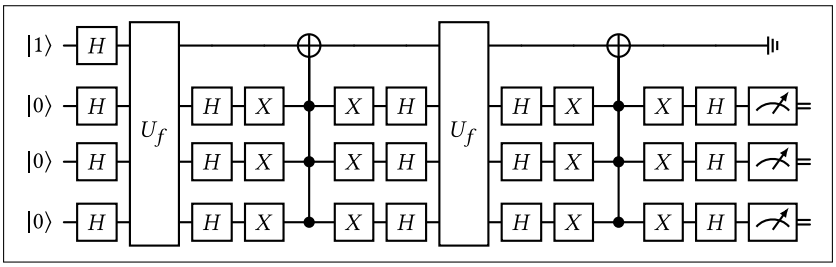

Fig. 23. The circuit built by the grover function applied to an oracle U_f of input size $n = 3$.

analyzing the size of the circuits built by one such program, we have to run it and then traverse the resulting circuit to compute its size, two operations that, although classical, can be costly, taking up to minutes on larger input sizes [31]. Furthermore, the resulting estimate is only valid for the specific input size we ran the program on.

Table 1. Performance of QuRA on the Test Quantum Programs

Program	LOC	Checking time (ms)	Width upper bound
`dumbNot`	10	86	2
`teleportation`	31	85	3
`qft`	23	96	n
`grover`	55	139	$\max(n+1, ow)$
`adder`	79	217	$3n+1$

Tests are run on an M1 MacBook Air with 16 GB of RAM.

On the other hand, if we rely on a *static* resource analysis tool such as QuRA, we can just analyze the program in a matter of milliseconds, without running it, and obtain a width upper bound that is valid for *all* input sizes. For example, the results that we obtained in Section 7.3.2 tell us that in order to safely add together two 32-qubit integers, we need a quantum computer with at least 97 logical qubits. If we switch to a 2,048-qubit addition, we need 6,145 logical qubits. If for some reason we need to add together two 10^6-qubit integers, we need $3 \times 10^6 + 1$ logical qubits, and so on and so forth. We obtain all of these estimates by just instantiating n in $3n + 1$ to the appropriate input size, without ever running the `adder` program. This is the real advantage of static analysis.

8 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-R [54] and going forward with Selinger and Rios's Proto-Quipper-M [52]. In the last 5 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* [11, 25, 42], available in the `Quipper` language, or dependent types [26]. Although some embryonic contributions in the direction of analyzing the size of circuits produced using `Quipper` have been given [63], 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 [26], 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. [6] and Liu et al. [43] 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. [17] 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. [34] and Yamakami's algebra of functions [67] in the style of Bellantoni and Cook [9], 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 [3, 37, 38] and term rewriting systems [4, 5, 48] to probabilistic [40] and object-oriented programs [23, 33]. 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 [22], which have been used extensively in the literature to automatically verify interesting properties of programs [44, 70]. In particular, the work of Vazou et al. on `Liquid Haskell` [64, 65] has been of particular inspiration, on account of `Quipper` being embedded precisely in `Haskell`. The liquid type system [53] 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 [35].

9 Conclusion and Future Work

In this article 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 realistic quantum algorithms, through the `QuRA` tool. 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.

9.1 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 *single* 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 we were instead interested in the *depth* of a circuit, then we would need a slightly different approach. 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*. 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. This last approach could also be used to keep track of the *fidelity* of qubits, if we allowed index terms to take on real values, although we consider this extension to lie well beyond the scope of this work.

Before moving on, it should be noted 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. Depending on the underlying technology, different quantum architectures face different bottlenecks, so operations that are deemed costly on one kind of quantum computer may be

inexpensive on another, and vice versa. For instance, one may want to keep track of the maximum number of *qubits* needed, ignoring the less critical number of classical bits, or to only focus on the number of *T-gates* required to describe a circuit, which is a common bottleneck in modern quantum architectures.

9.2 Further Work

Among other topics for further work, we can identify two main research directions. On the 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 [25, 26, 42, 52]. 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 9.1 we briefly discussed how our system could be modified to handle the analysis of different resource types. It would be interesting to pursue this path both theoretically and in the implementation of `QuRA`. If we could *actually* generalize our resource analysis, that is, make it parametric on the kind of resource being analyzed, we would be able to analyze the same program in the same language under different resource usage metrics, in a very flexible fashion.

References

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

[2] Parfait Atchade-Adelomou and Saul Gonzalez. 2023. Efficient quantum modular arithmetics for the ISQ era. arXiv:2311.08555. Retrieved from https://arxiv.org/abs/2311.08555

[3] Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. of ICFP 2015. ACM*, 152–164. DOI: https://doi.org/10.1145/2784731.2784753

[4] Martin Avanzini and Georg Moser. 2008. Complexity analysis by rewriting. In *Proc. of FLOPS 2008*. Springer, Berlin, 130–146.

[5] Martin Avanzini and Georg Moser. 2013. Tyrolean complexity tool: Features and usage. In *Proc. of RTA 2013*, Vol. 21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 71–80. DOI: https://doi.org/10.4230/LIPIcs.RTA.2013.71

[6] Martin Avanzini, Georg Moser, Romain Pechoux, Simon Perdrix, and Vladimir Zamdzhiev. 2022. Quantum expectation transformers for cost analysis. In *Proc. of LICS 2022*. ACM. DOI: https://doi.org/10.1145/3531130.3533332

[7] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Dana Fisman and Grigore Rosu (Eds.), Springer International Publishing, Cham, 415–442.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Retrieved from www.SMT-LIB.org

[9] Stephen Bellantoni and Stephen Cook. 1992. A new recursion—Theoretic characterization of the polytime functions (extended abstract). In *Proc. of STOC 1992*. ACM, 283–293. DOI: https://doi.org/10.1145/129712.129740

[10] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2021. *Handbook of Satisfiability* (2nd ed.). Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. DOI: https://doi.org/10.3233/FAIA336

[11] Andrea Colledan and Ugo Dal Lago. 2023. On dynamic lifting and effect typing in circuit description languages. In *Proc. of TYPES 2022*, Vol. 269. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 3:1–3:21. DOI: https://doi.org/10.4230/LIPIcs.TYPES.2022.3

[12] Hugh Collins and Chris Nay. 2022. IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum System Two. Retrieved June 6, 2025 from https://is.gd/WPV7lO

[13] Emily Conover. 2020. Light-Based Quantum Computer Jiuzhang Achieves Quantum Supremacy. Retrieved June 6, 2025 from https://is.gd/zIgFzK

[14] D. Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring, IBM research report RC19642. arXiv:quant-ph/0201067. Retrieved from https://arxiv.org/abs/quant-ph/0201067

[15] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, et al. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (Sep. 2022), 50 pages. DOI: https://doi.org/10.1145/3505636

[16] Ugo Dal Lago and Marco Gaboardi. 2011. Linear dependent types and relative completeness. In *Proc. of LICS 2011*. IEEE Computer Society, 133–142. DOI: https://doi.org/10.1109/LICS.2011.22

[17] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. 2010. Quantum implicit computational complexity. *Theoretical Computer Science* 411, 2 (2010), 377–409. DOI: https://doi.org/10.1016/j.tcs.2009.07.045

[18] Ugo Dal Lago and Barbara Petit. 2012. Linear dependent types in a call-by-value scenario. In *Proc. of PPDP 2012*. ACM, 115–126. DOI: https://doi.org/10.1145/2370776.2370792

[19] Ugo Dal Lago and Barbara Petit. 2013. The geometry of types. In *Proc. of POPL 2013*. ACM, 167–178. DOI: https://doi.org/10.1145/2429069.2429090

[20] Richard Eisenberg, Alexander Green, Peter Lumsdaine, Keith Kim, Siun-Chuon Mau, Baranidharan Mohan, Won Ng, Joel Ravelomanantsoa-Ratsimihah, Neil Ross, Artur Scherer, et al. 2019. Quipper.Libraries.QFT. Retrieved June 6, 2025 from https://is.gd/AEJmp9

[21] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. 2018. Open source software in quantum computing. *PLoS One* 13, 12 (2018), e0208561. DOI: https://doi.org/10.1371/journal.pone.0208561

[22] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proc. of PLDI 1991*. ACM, 268–277. DOI: https://doi.org/10.1145/113445.113468

[23] Florian Frohn and Jürgen Giesl. 2017. Complexity analysis for Java with AProVE. In *Proc. of IFM 2017*. Springer International Publishing, 85–101.

[24] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2020. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In *Proc. of RC*. Springer International Publishing, 153–168.

[25] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2023. Proto-Quipper with dynamic lifting. In *Proc. of POPL 2023*. ACM. DOI: https://doi.org/10.1145/3571204

[26] Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear dependent type theory for quantum programming languages: Extended abstract. In *Proc. of LICS 2020. ACM*, 440–453. DOI: https://doi.org/10.1145/3373718.3394765

[27] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proc. of POPL 2013. ACM*, 357–370. DOI: https://doi.org/10.1145/2429069.2429113

[28] Simon J. Gay. 2006. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science* 16, 4 (2006), 581–600. DOI: https://doi.org/10.1017/S0960129506005378

[29] Jerry D. Gibson. 2023. *Fourier Transforms, Filtering, Probability and Random Processes: Introduction to Communication Systems*. Springer International Publishing. DOI: https://doi.org/10.1007/978-3-031-19580-8

[30] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. An introduction to quantum programming in Quipper. In *Proc. of RC*. Springer, Berlin, 110–124.

[31] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Valiron Benoît. 2013. Quipper. In *Proc. of PLDI. ACM*, 333–342. DOI: https://doi.org/10.1145/2499370.2462177

[32] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. arXiv:quant-ph/9605043. Retrieved from https://arxiv.org/abs/quant-ph/9605043

[33] Emmanuel Hainry and Romain Péchoux. 2013. Type-Based Heap and Stack Space Analysis in Java. Technical Report.

[34] Emmanuel Hainry, Romain Péchoux, and Mário Silva. 2023. A programming language characterizing quantum polynomial time. In *Proc. of FoSSaCS 2023*. Springer-Verlag, 156–175. DOI: https://doi.org/10.1007/978-3-031-30829-1

[35] Martin Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate your assets: Reasoning about resource usage in liquid Haskell. *PACMPL* 4 (2019), 1–27. DOI: https://doi.org/10.1145/3371092

[36] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum algorithm for linear systems of equations. *Physical Review Letters* 103, 15 (2009), 150502. DOI: https://doi.org/10.1103/PhysRevLett.103.150502

[37] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource aware ML. In *Computer Aided Verification*. P. Madhusudan and Sanjit A. Seshia (Eds.), Springer, Berlin, 781–786.

[38] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential. In *Programming Languages and Systems*. Andrew D. Gordon (Ed.), Springer, Berlin, 287–306.

[39] Ranjit Jhala and Niki Vazou. 2020. Refinement types: A tutorial. arXiv:2010.07763. Retrieved from https://arxiv.org/abs/2010.07763

[40] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest precondition reasoning for expected run-times of probabilistic programs. In *Programming Languages and Systems*. Peter Thiemann (Ed.), Springer, Berlin, 364–389.

[41] E. Knill. 2022. Conventions for quantum pseudocode. arXiv:2211.02559. Retrieved from https://arxiv.org/abs/2211.02559

[42] Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. 2021. Concrete categorical model of a quantum circuit description language with measurement. In *Proc. of FSTTCS*, 51:1–51:20. DOI: https://doi.org/10.4230/LIPIcs.FSTTCS.2021.51

[43] Junyi Liu, Li Zhou, Gilles Barthe, and Mingsheng Ying. 2022. Quantum weakest preconditions for reasoning about expected runtimes of quantum programs. In *Proc. of LICS 2022*. ACM, Article 4, 13 pages. DOI: https://doi.org/10.1145/3531130.3533327

[44] Yitzhak Mandelbaum, David Walker, and Robert Harper. 2003. An effective theory of type refinements. In *Proc. of ICFP 2003*. ACM Press, 213–225. DOI: https://doi.org/10.1145/944705.944725

[45] John Martinis. 2019. Quantum Supremacy Using a Programmable Superconducting Processor. Retrieved June 6, 2025 from https://is.gd/v3VXFi

[46] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information* (10th Anniversary ed.). Cambridge University Press. DOI: https://doi.org/10.1017/CBO9780511976667

[47] Flemming Nielson and Hanne Riis Nielson. 1999. Type and effect systems. In *Correct System Design: Recent Insights and Advances*. Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Springer, Berlin, 114–136. DOI: https://doi.org/10.1007/3-540-48092-7

[48] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning* 51, 1 (2013), 27–56. DOI: https://doi.org/10.1007/s10817-013-9277-6

[49] Jens Palsberg. 2019. Toward a universal quantum programming language. *XRDS: Crossroads, The ACM Magazine for Students* 26, 1 (2019), 14–17. DOI: https://doi.org/10.1145/3355759

[50] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A core language for quantum circuits. In *Proc. of POPL 2017. ACM*, 846–858. DOI: https://doi.org/10.1145/3009837.3009894

[51] John Preskill. 2012. Quantum computing and the entanglement frontier. arXiv:1203.5813. Retrieved from https://arxiv.org/abs/1203.5813

[52] Francisco Rios and Peter Selinger. 2017. A categorical model for a quantum circuit description language. In *Proc. of QPL 2017*, 164–178. DOI: https://doi.org/10.4204/EPTCS.266.11

[53] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proc. of PLDI 2008. ACM*, 159–169. DOI: https://doi.org/10.1145/1375581.1375602

[54] Neil Ross. 2015. Algebraic and Logical Methods in Quantum Computation. Ph.D. Dissertation. Dalhousie University.

[55] J. W. Sanders and P. Zuliani. 2000. Quantum programming. In *Proc. of MPC 2000*. Springer, Berlin, 80–99.

[56] Maximilian Schlosshauer. 2007. *Decoherence and the Quantum-to-Classical Transition*. Springer, Berlin. DOI: https://doi.org/10.1007/978-3-540-35775-9

[57] Peter Selinger. 2004. A brief survey of quantum programming languages. In *Proc. of FLOPS 2004*, 1–6.

[58] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586. DOI: https://doi.org/10.1017/S0960129504004256

[59] Peter Selinger and Benoît Valiron. 2005. A lambda calculus for quantum computation with classical control. In *Proc. of TLCA*. Springer, Berlin, 354–368. DOI: https://doi.org/10.1007/11417170

[60] P. W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. of FOCS 1994*. IEEE Computer Society Press. DOI: https://doi.org/10.1109/sfcs.1994.365700

[61] Lau Skorstengaard. 2019. An introduction to logical relations. arXiv:1907.11133. Retrieved from https://arxiv.org/abs/1907.11133

[62] Thomas Sturm. 2017. A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Mathematics in Computer Science* 11, 3–4 (Apr. 2017), 483–502. DOI: https://doi.org/10.1007/s11786-017-0319-z

[63] Benoit Valiron. 2016. Automated, Parametric Gate Count of Quantum Programs. DOI: https://doi.org/10.14288/1.0319340

[64] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with refinement types in the real world. In *Proc. of Haskell 2014. ACM*, 39–51. DOI: https://doi.org/10.1145/2633357.2633366

[65] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proc. of ICFP 2014*. ACM, 269–282. DOI: https://doi.org/10.1145/2628136.2628161

[66] Vlatko Vedral, Adriano Barenco, and Artur Ekert. 1996. Quantum networks for elementary arithmetic operations. *Physical Review. A, Atomic, Molecular, and Optical Physics* 54, 1 (Jul. 1996), 147–153. DOI: https://doi.org/10.1103/PhysRevA.54.147

[67] Tomoyuki Yamakami. 2020. A schematic definition of quantum polynomial time computability. *The Journal of Symbolic Logic* 85, 4 (2020), 1546–1587. DOI: https://doi.org/10.1017/jsl.2020.45

[68] NosonS. Yanofsky and Mirco A. Mannucci. 2008. *Quantum Computing for Computer Scientists* (1st ed.). Cambridge University Press.

[69] Mingsheng Ying. 2016. *Foundations of Quantum Programming* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA.

[70] Ezgi Çiçek, Deepak Garg, and Umut Acar. 2015. Refinement types for incremental computational complexity. In *Programming Languages and Systems*. Jan Vitek (Ed.), Vol. 9032. Springer, Berlin, 406–431. DOI: https://doi.org/10.1007/978-3-662-46669-8

[71] Bernhard Ömer. 2005. Classical concepts in quantum programming. *International Journal of Theoretical Physics* 44, 7 (2005), 943–955. DOI: https://doi.org/10.1007/s10773-005-7071-x

Received 26 April 2024; revised 6 May 2025; accepted 13 May 2025