Flexible Type-Based Resource Estimation in Quantum Circuit Description Languages

ANDREA COLLEDAN, University of Bologna, Italy and INRIA Sophia Antipolis, France
UGO DAL LAGO, University of Bologna, Italy and INRIA Sophia Antipolis, France

We introduce a type system for the `Quipper` language designed to derive upper bounds on the size of the circuits produced by the typed program. This size can be measured according to various metrics, including *width*, *depth* and *gate count*, but also variations thereof obtained by considering only *some* wire types or *some* gate kinds. The key ingredients for achieving this level of flexibility are effects and refinement types, both relying on *indices*, that is, generic arithmetic expressions whose operators are interpreted differently depending on the target metric. The approach is shown to be correct through logical predicates, under reasonable assumptions about the chosen resource metric. This approach is empirically evaluated through the `QuRA` tool, showing that, in many cases, inferring tight bounds is possible in a fully automatic way.

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

Additional Key Words and Phrases: Effects, Refinement Types, Lambda Calculus, Quantum Computing, Quipper

ACM Reference Format:
Andrea Colledan and Ugo Dal Lago. 2025. Flexible Type-Based Resource Estimation in Quantum Circuit Description Languages. *Proc. ACM Program. Lang.* 9, POPL, Article 47 (January 2025), 31 pages. https://doi.org/10.1145/3704883

1 Introduction

Since its introduction, the quantum computing paradigm has promised to have a disruptive impact on many areas of computing, ranging from cryptography [56, 62] to machine learning [67]. If, on the one hand, these promises have found clear confirmation on the side of the underlying computational model, i.e. that of quantum circuits, on the other hand the implementation of quantum circuits with acceptable error rates is still considered a difficult task [28, 55], even when the amount of quantum data or the number of operations to be executed is relatively small. Despite the great progress recorded in the last five years [9, 10, 51], it seems safe to say that qubits and the ability to perform quantum operations on them are, unlike their classical relatives, very precious resources, which need to be accounted for with great care: as soon as the number of qubits gets nontrivial, coherence problems start to arise [60], and the very number of operations that one can perform on qubits before their state becomes too noisy is very low.

Quantum architectures are thus intrinsically more error-prone than their classical counterparts. In spite of this, they are still accessed through high-level programming languages, and proposals in this direction abound [27, 61]. Most quantum programming languages are somehow hybrid

Authors' Contact Information: Andrea Colledan, andrea.colledan@unibo.it, University of Bologna, Bologna, Italy and INRIA Sophia Antipolis, Valbonne, France; Ugo Dal Lago, ugo.dallago@unibo.it, University of Bologna, Bologna, Italy and INRIA Sophia Antipolis, Valbonne, France.

ACM 2475-1421/2025/1-ART47
https://doi.org/10.1145/3704883

and allow *both* classical *and* quantum computing to happen within the same program. It is thus the job of the compiler, or of the interpreter, or of the very programmer, to split the workload between classical, slow, but fault-tolerant hardware and quantum, fast, but faulty devices. Some of the existing programming languages are designed around the QRAM model [44], in which control is classical and data are quantum. Others, instead, take the form of *circuit description languages*, that is, programming languages and libraries in which programs are not meant to directly execute quantum operations, but rather implement circuit-building algorithms, whose output circuits represent the quantum computations later executed on a quantum device.

In both QRAM-based and circuit description languages, having a precise idea of how many qubits and how many quantum operations are needed to solve a generic instance of a computational problem is of paramount importance, and the corresponding verification problem has recently received the attention of the programming language community [8, 50]. One might imagine this kind of analysis to closely resemble the classical counterpart, and rightly so. However, some key differences are at play. Of course, the underlying computational model is different, and this needs to be taken into account. But there is more: even if we focus on the quantum circuit model only, existing hardware architectures are anything but homogeneous, and different devices have their own cost-related strengths and weaknesses. For example, in some architectures the operation of swapping between qubits is extremely expensive, while in others it is not. As another example, circuits often contain both classical (bit) and quantum (qubit) wires, and while keeping the latter under control is of paramount importance, the same cannot be said about the former. In other words, any resource analysis framework for quantum circuits should be designed so as to be *as flexible as possible*, accomodating for different resource metrics and for varying definitions of the same metric.

In this paper we go precisely towards a flexible and scalable methodology for quantum resource estimation, by defining a family of type systems for the `Quipper` language where each type system is capable of deriving upper bounds on the size of the circuits produced by a program with respect to a specific metric. In fact, all of these type systems can be seen as instances of *one* type system, called Proto-Quipper-RA, in which some operators occurring in types and dealing with resource estimation are left uninterpreted. The aforementioned family is thus indexed on a *resource metric interpretation* (RMI), that is, an interpretation of these symbols. Remarkably, an instance of Proto-Quipper-RA is provably correct with respect to the chosen metric once certain reasonable inequalities between the RMI and the metric's ground truth are shown to hold. When coming up with our approach, we took inspiration from our previous work on width estimation [8], substantially generalizing it in order to ensure that a wide range of circuit metrics can be dealt with: not only *global* metrics such as width, but also metrics that are by their nature *local* to wires, such as the depth of a circuit.

Technically, Proto-Quipper-RA is based on two fundamental ingredients, namely refinement types [23, 65] and effects [52]. The former allow us to attach relevant quantitative information to wire types, i.e. types that represent data in the underlying quantum circuit, and thus allow for local metrics, such as depth, to be treated. Effects, on the other hand, attribute to each circuit-building computation a description of the size of the resulting circuit *as a whole*, and as such are suitable for managing global metrics, as already shown in [8]. What ties the two types of metrics together as the vehicle for expressing bounds is a language of *index terms*, i.e. arithmetic expressions that contain natural-valued variables, and as such can express upper bounds that depend on circuit-building parameters. Index terms are well-known to provide a flexible and powerful methodology for injecting arithmetic reasoning within a type system [13, 16, 17, 26].

The key contributions of this work are the following:

- On the one hand, the proof that the definitional apparatus based on Proto-Quipper-RA and RMIs is, under mild assumptions, correct by construction. In other words, that the derivable type judgments indeed provide correct upper bounds to the corresponding circuit metric. The proof proceeds by adapting the logical predicates technique to Proto-Quipper-RA.
- On the other hand, the implementation of the generic Proto-Quipper-RA framework as the core of `QuRA`, a tool for the resource verification of circuit description languages.[1] By bringing Proto-Quipper-RA's genericity to `QuRA`, we were able to effortlessly extend its functionality to the verification of gate count, depth, and several variations of these and other circuit metrics.

The rest of the paper is structured as follows. Section 2 introduces some core concepts underpinning this work: quantum circuits, their size, `Quipper` and the problem of estimating the size of the circuits built by circuit description languages. Section 3 then provides a brief, informal introduction to the Proto-Quipper family of calculi [8, 57, 58], which our formalism is based on. Section 4 picks up where Section 2 left off, formalizing the notions of circuit and circuit metric and introducing a simple parametric type system for the estimation of the size of individual circuits. Section 5 raises the level of abstraction, introducing Proto-Quipper-RA, a circuit description language with a type system capable of estimating different circuit metrics depending on an underlying semantic interpretation of resources. Proto-Quipper-RA is proven to be correct under mild assumptions in Section 6. Section 7 describes the efforts to implement Proto-Quipper-RA within `QuRA`, and shows that the latter can be used to analyze real-world quantum algorithms both in terms of global and local resource consumption. Lastly, sections 8 and 9 present the related literature, conclusions, and future work.

2 Resource Estimation in Circuit Description Languages: An Overview

This section is dedicated to giving an overview of the resource estimation problem in quantum circuit description languages, and particularly in `Quipper`. Programs written in these languages use both classical resources, necessary for the *construction* of a quantum circuit, and quantum resources, necessary for its *execution*. Given their scarcity, quantum resources are the ones we are interested in keeping under control. These means giving bounds on the *size* of circuits. But what do we mean by that?

2.1 Quantum Circuits and Their Size

Quantum circuits are the standard computational model of quantum computation. A circuit consists of an ordered sequence of *gates* applied to a collection of *wires*. The latter represent the individual bits and qubits involved in a computation, while gates represent the elementary operations applied to them, such as unitary transformations, measurements, initializations, and so on. Quantum circuits are often represented graphically, like in Figure 1: time flows from left to right, horizontal lines represent either qubit (when single) or bit (when double) wires, and the various symbols that lie on them represent different operations applied to the corresponding qubits or bits. Any wire starting from $|\phi\rangle$ is initialized as part of the computation to the known state ϕ, while all other wires are input to the circuit. Any wire that ends with the ground symbol is discarded as part of the computation, while the others contain the circuit's output.

When faced with the question "what is the size of a circuit", there are many ways in which one can answer. One can first of all consider a measure of *space*, namely the *width* of the circuit. This is defined as the maximum number of wires active at any point in time. By giving an upper bound n to the width of a circuit, we can be sure that no more than n qubits and bits are necessary to

[1] https://github.com/andreacolledan/qura

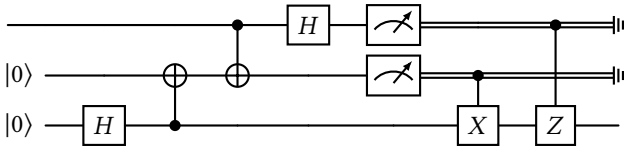

Fig. 1. An example of quantum circuit: the quantum teleportation circuit

execute the circuit, i.e. that any hardware architecture providing at least n logical bits and qubits can evaluate the circuit on any given input. What if we are rather interested in keeping track of *time*? In this case it makes more sense to consider the *depth* of the circuit, namely the maximum number of gates encountered on any path from an input wire or a wire creation to an output wire or a wire discarding. The resulting metric nicely captures a form of parallel running time. Keeping the depth under control is helpful in all those scenarios in which quantum data get progressively less reliable as time passes, or as more operations are applied to them. A third complexity measure, somehow a mixture of the previous two, is the so called *gate count* of a circuit, simply defined as the total number of gates occurring in it. This gives a measure of the total number of operations applied to the quantum and classical data during the execution of the circuit. For example, the circuit in Figure 1 has a gate count of 8, a width of 3 and a depth of 6. Note that, as a convention, initializations and discardings do not count towards the gate count and depth calculations, while measurements do.

2.2 From Circuits to `Quipper`

Quantum circuits can be described gate-by-gate, either graphically, like we did in Figure 1, or with the use of specific languages (like `QASM` [12]). However, because they have fixed input size, individual quantum circuits cannot truly implement algorithms (such as Shor's [62] or Grover's [32]), which are by definition capable of dealing with arbitrarily large inputs. Rather, quantum algorithms are represented by *circuit families*, which contain a different circuit for each input size. But how do we describe circuit families? This is where circuit description languages come into play: by manipulating already built circuits and combining them together in a way that *depends* on the value of one or more (classical) parameters, a single circuit description program can construct an entire family of circuits, and thus implement an algorithm. This approach to circuit description is nowadays adopted by many high-level quantum circuit description languages and libraries, such as `Qiskit` [41], `Cirq` [22] and `Quipper` [30].

`Quipper`, in particular, allows programmers to describe quantum circuits in a very high-level fashion. Besides the gate-by-gate approach, `Quipper` also supports parametric and hierarchical circuits, promoting a view in which circuits are first-class citizens. `Quipper` has been shown to be scalable, that is to say, it has been shown to be able to describe complex quantum algorithms that, depending on input parameters, translate to circuits possibly involving trillions of gates applied to millions of qubits [31].

Our previous work [8] introduces a novel methodology by which `Quipper` programs can be statically analyzed as for the width of the circuits they produce, all this parametrically on the input size. However, notions of size like depth and gate count simply *cannot* be dealt with in this framework. Moreover, the considered notion of width cannot be tuned in any way (e.g. so as to ignore bits, which are less critical). In a sense, what we do in this paper can be seen as a nontrivial generalization of the techniques in [8], in which not only depth and gate count, but in fact *any* metric satisfying certain simple conditions can be analyzed.

```
1  dumbNot :: Qubit -> Circ Qubit
2  dumbNot q = do
3    a <- qinit True
4    (q,a) <- controlled_not q a
5    qdiscard a
6    return q
```

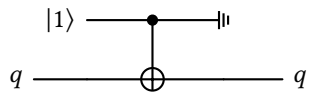

Fig. 2. A Quipper function implementing quantum negation, and the circuit it builds

```
1  iter :: (Qubit -> Circ Qubit) -> Int
2            -> Qubit -> Circ Qubit
3  iter f 0 q = return q
4  iter f n q = do
5    q <- f q
6    iter f (n-1) q
```

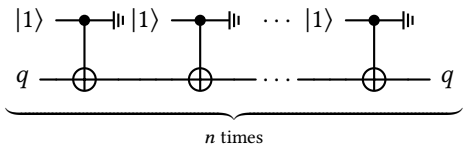

Fig. 3. A higher-order Quipper function and the result of its application to dumbNot from Figure 2

This section showcases a number of Quipper programs, in order to illustrate how refinements and effects can be used to keep track of the resource information that we are interested in. In doing so, we will find that width and gate count have to be treated in a fundamentally different way than depth. We avoid any formality for now, leaving it for the forthcoming sections.

Let us start with the example of Figure 2. The Quipper function on the left builds the quantum circuit on the right: a dumb and expensive 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 qubit does not appear in the interface of the circuit, but it clearly has an impact on its overall size. Recalling what we said previously, we easily see that this circuit has width 2, gate count 1 and depth 1.

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. Knowing the size of the circuit built by dumbNot, what is the size of the circuit built by `iter dumbNot n`? From a width perspective, it is easy to see that the composition in sequence obeys some form of *maximum* semantics: outputs become inputs and discarded wires are reused when new ones are initialized. Thus, the resulting circuit has still width 2. What about its gate count and depth? From their perspective, the composition in sequence clearly obeys a form of *addition* semantics, although for different reasons: in the first case, the gate count of two circuits composed together in any way is the sum of their individual gate counts, whereas in the case of depth, two circuits stringed together on the same path have depth equal to the sum of their indivdual depths. Therefore, the circuit built by `iter dumbNot n` has gate count and depth both equal to n.

2.2.1 A Type-Based Analysis. Intuitively, this kind of reasoning seems natural, but how can we draw the same conclusions by means of static analysis? We are facing two major hurdles here: higher-order types and input-dependent size. As for the former, we closely follow the approach in [8]. In the cited paper, the arrow type is annotated both with the size of the circuit produced by the corresponding function once applied (a standard technique in effect typing [53]) and with information about the wires enclosed in the function's closure (in a way reminiscent of closure types [59]). We thus work with arrows of the form $A \to^{I}_{T} B$, where I is an expression representing the size of the circuit produced by the function, and T is a composite type representing what

```
1  mapHadamard :: [Qubit] -> Circ [Qubit]
2  mapHadamard [] = return []
3  mapHadamard (q:qs) = do
4    q <- hadamard  q
5    qs <- mapHadamard qs
6    return (q:qs)
```

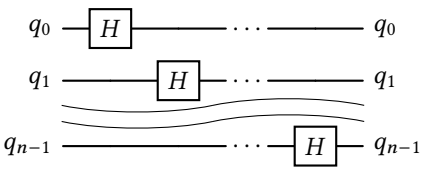

Fig. 4. A Quipper function with variable input size, and the circuit it builds on an input of size n

wires are captured by the function. If no wires are captured, T is the unit type $\mathbb{1}$. This last piece of information is essential to guarantee the correctness of the analysis even in the case of data hiding, since wires have a size even when not used. Thus, depending on the resource being considered, the dumbNot function has type either Qubit $\to^2_{\mathbb{1}}$ Qubit (for width) or Qubit $\to^1_{\mathbb{1}}$ Qubit (for gate count and depth). This brings us to input dependency. It is clear that the aforementioned size annotation cannot be a mere *constant*, but should rather *depend* on the parameters of the circuit-building function, in this case the number n that we pass to `iter`. We solve this problem by allowing size annotations to contain natural-valued variables, which correspond to these classical parameters. All in all, knowing that dumbNot has width-annotated type Qubit $\to^2_{\mathbb{1}}$ Qubit, the system is capable of correctly attributing type Qubit $\to^2_{\mathbb{1}}$ Qubit to the `iter dumbNot n` term, by trivially taking the maximum of n copies of 2. Similarly, knowing that dumbNot has gate-count-annotated type Qubit $\to^1_{\mathbb{1}}$ Qubit, the system is capable of giving type Qubit $\to^n_{\mathbb{1}}$ Qubit to `iter dumbNot n`, by adding up n copies of 1. The same goes for depth.

Let us now consider a slightly different example. Figure 4 shows a Quipper function that returns a circuit in which the Hadamard gate is applied to n qubits, a common preprocessing step in many quantum algorithms. Knowing that by itself the Hadamard gate has width, gate count and depth equal to 1, what size is the circuit produced by mapHadamard when we apply it to a list of n qubits? This is slightly more complicated than the previous example: at each step we apply a single Hadamard gate, while $n-1$ qubits flow alongside it. As we mentioned earlier, wires have a size of their own. Specifically, each wire has width equal to 1, and depth and gate count trivially equal to 0. A bundle of wires of width $n-1$ thus flows alongside each Hadamard gate. This is to all effects a composition *in parallel* of two circuits, even though one of them is just wires. How do we behave in this scenario? From the point of view of width, composition in parallel clearly obeys a sum semantics. We therefore know that each step of mapHadamard has width n, gate count 1 and depth 1. If we follow a purely compositional approach, like in the previous example, by iterating n steps in sequence we conclude that mapHadamard applied to a list of n qubits produces a circuit of width n, gate count n *and depth* n. Using the same notation for sized lists adopted in [8], we can write the type of mapHadamard as List^n Qubit $\to^n_{\mathbb{1}}$ List^n Qubit.

2.2.2 Global and Local Metrics. Immediately, it is clear that something is not quite right: while we got exact estimates for width and gate count, the estimate for depth is a gross overapproximation. Recall that depth is defined as the maximum number of gates encountered *on any path* from an input to an output of the circuit. It is obvious that every such path in the circuit in Figure 4 has only 1 gate on it, regardless of n. However, an overapproximation arises because we have abstracted away information about *which output wires* are matched to *which input wires*, so that when we sequence the n steps, we must assume the worst case scenario, i.e. that the operations of each step are applied to the same wire.

This example highlights a fundamental distinction between metrics such as width and gate count and metrics such as depth. The former are in a way properties of the circuit as a whole, they are

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	$::= \mathbb{1} \mid w \mid T \otimes U$
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	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\,M \mid (\bar{\ell}, C, \bar{k}) \mid \langle V, W\rangle$ $\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}\ V\ W\ X$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle$

Fig. 5. Types and syntax of Proto-Quipper

global and thus amenable to good notions of composition in sequence and parallel. On the other hand, depth is a property of *wires*, rather than *circuits*, and can only be composed adequately at the scale of the individual qubit or bit. The traditional definition of depth of a circuit is thus just an aggregation of the depth of its wires. This observation motivates the most original contribution of this work: instead of keeping track of depth (or other local metrics) via effects, like we do with width and gate count, we do so by directly *refining wire types with depth information*. Let us momentarily ignore global metrics. Let Qubit^i be the type of a qubit wire at depth i and let `hadamard` have type $\mathsf{Qubit}^i \to \mathsf{Qubit}^{i+1}$, meaning that the application of a Hadamard gate to a qubit increases its depth by 1. Then, we can easily derive that `mapHadamard` has type $\mathsf{List}^n\,\mathsf{Qubit}^i \to \mathsf{List}^n\,\mathsf{Qubit}^{i+1}$. If we choose $i = 0$ and take the max of n copies of $i + 1 = 1$, we correctly conclude that the depth of the circuit built by the function is 1.

3 The Proto-Quipper Language

As a language, `Quipper` is embedded in `Haskell`, and because of this it lacks a proper formal semantics. However, over the years, a number of calculi have been developed in order to formalize fragments [57, 58] and extensions [7, 24, 25, 46] of the `Quipper` language. These calculi are referred to collectively as the Proto-Quipper language family. In its most basic form, Proto-Quipper is a linear lambda calculus equipped with bespoke constructs to build circuits. Circuits are built as a side-effect of a computation, behind the scenes, but more importantly they can also be manipulated as data within the language.

The grammars for Proto-Quipper types and terms are given in Figure 5. Speaking at a high level, we can say that Proto-Quipper employs a linear-nonlinear typing discipline. In particular, wire types $w \in \{\mathsf{Qubit}, \mathsf{Bit}\}$ and arrows are treated linearly. A subset of types, called *parameter types*, represents the values of the language that can be copied. As customary in linear logic, any term of type A can be *lifted* into a parameter type $!\,A$ provided it does not consume linear resources.

Let us focus on the language of terms, and specifically on its domain-specific constructs. On the side of values, we have *labels* and *boxed circuits*. A label ℓ represents a reference to an output wire of the circuit currently being built and it is unsurprisingly assigned a wire type. Ordered structures of labels form a subset of values called *wire bundles*, which are given *bundle types*. On the other hand, a boxed circuit $(\bar{\ell}, C, \bar{k})$ represents a circuit C as a datum within the language, together with its input and output wires, given as wire bundles $\bar{\ell}$ and $\bar{k}$. A boxed circuit is given type $\mathsf{Circ}(T, U)$, where T and U are respectively the input and output types of the circuit. Boxed circuits can be duplicated and manipulated (e.g. reversed) by primitive functions, but more importantly they can be appended to the underlying circuit via the apply operator. This is precisely how circuits are built in Proto-Quipper: the apply operator takes as first argument a boxed circuit $(\bar{\ell}, C, \bar{k})$ and appends

C to the underlying circuit $\mathcal{D}$. The second argument of apply, a bundle of wires $\bar{t}$ coming from the free output wires of $\mathcal{D}$, tells us *where exactly* to append C to $\mathcal{D}$.

We expect the language to be equipped with constant boxed circuits corresponding to fundamental circuit operations (e.g. Hadamard, measurement, etc.), but the programmer can also define their own custom circuits via the box operator. Intuitively, box takes as input a circuit-building function f and executes it in a sandboxed environment, isolated from the current circuit. Therefore, f produces a standalone circuit C, which is returned by box as a boxed circuit of the form $(\bar{\ell}, C, \bar{k})$.

On the classical side of things, note that Proto-Quipper does *not* support general recursion. Instead, a limited form of recursion on lists is provided in the form of a primitive fold construct, which takes as argument a duplicable step function V of type $!((B \otimes A) \multimap B)$, an initial accumulator W of type B, and folds V over a list X of type List A, to obtain a value of type B. Folds are not sufficient to recover the full power of general recursion, but they are expressive enough in the context of a circuit description language: quantum circuits are finite objects, so when we are building them we use iteration almost exclusively to apply some operation to each of the qubits in a list, or to initialize a known number of wires. This is a form of *bounded* iteration which is fully within the expressiveness of a fold construct.

To conclude this section, we just remark how all of the `Quipper` programs shown in Section 2 can easily be encoded in Proto-Quipper. However, Proto-Quipper's system of simple types is not meant to reason about resources and it is therefore unable to tell us anything about the size of the circuits produced by these programs. Of course, the option of implementing `mapHadamard`, running it on a concrete input parameter and checking the size of the circuit produced at run-time is not ruled out, but this approach amounts to *testing*, rather than *verification*, and it is unable to yield general, parametric results on the size of the circuits produced by circuit families, i.e. algorithms.

4 Circuits and Circuit Resource Metrics

So far, we have talked about circuits and their size metrics very informally, at a high level. This sections aims to formalize these two concepts, starting with what a *quantum circuit* is and proceeding to outline the class of metrics we are interested in analyzing.

4.1 The Circuit Representation Language

In the literature [25, 46, 57], circuits are usually taken to be morphisms in some symmetric monoidal category. This model for circuits has the advantage of being abstract and highly compositional, but it is not particularly handy for reasoning about *intensional* properties of circuits, such as their size. For this reason, we adopt instead the concrete circuit model presented in [8], called *Circuit Representation Language* (CRL) as the target of our notion of circuit description. As we said, individual circuits can be described in a gate-by-gate fashion, which is exactly what is done in CRL:

$$CRL \quad C, \mathcal{D}, \mathcal{E}, \mathcal{F} := id_Q \mid C; g(\bar{\ell}) \to \bar{k}, \tag{1}$$

Here Q is a collection of typed wires, $\bar{\ell}, \bar{k}$ are wire bundles as introduced in Section 3, and g comes from an arbitrary, but fixed set $\mathscr{G}$ of elementary operations. In other words, a circuit is either the identity on a collection of wires Q or a circuit followed by the application of operation g to the wires in $\bar{\ell}$. Operation g outputs the wires in $\bar{k}$. Note that we assume $\mathscr{G}$ to include both quantum gates and more general operations such as measurements, wire initializations, etc. Also, circuits can be concatenated: we write $C :: \mathcal{D}$ to denote C, followed by all the elementary operations in $\mathcal{D}$.

4.2 Recursive Circuit Metrics

Given a CRL circuit, how do we measure its size? That is, what is a *circuit metric*? In general, we are interested in metrics that are defined recursively on the structure of a circuit, and this leads us to the following definition.

Definition 4.1 (Recursive Circuit Metric). A *recursive circuit metric* $\mathscr{R}$ is a pair $(A, \mu_{\mathscr{R}})$, where A is a set called the *carrier* of $\mathscr{R}$ and $\mu : CRL \to A$ is the *metric function* of $\mathscr{R}$, defined by structural recursion on circuits.

In the wake of Section 2, we distinguish between two fundamental classes of recursive circuit metrics: *global* and *local* metrics. Global metrics assign a single natural quantity to an entire circuit and in a way formalize the notion of *size* of a circuit. Thus, we say that a recursive circuit metric is *global* when $A = \mathbb{N}$. Local metrics, on the other hand, assign a natural quantity to each *wire* of a circuit. Thus, a local circuit metric has carrier $A = (\mathscr{L} \to \mathbb{N}) \to (\mathscr{L} \to \mathbb{N})$, where $\mathscr{L}$ is the set of wire labels. This signature means that a local metric assigns to each circuit a higher-order function that, given an assignment of resource values to the *input* wires of a circuit, returns an assignment of resource values to the *output* wires of the circuit. The following examples illustrate how some common circuit metrics fall into this categorization.

Example 4.2 (Global Metrics). The circuit metrics **GCount** $= (\mathbb{N}, gatecount)$ and **Width** $= (\mathbb{N}, width)$, where

$$gatecount(id_Q) = 0, \qquad gatecount(C; g(\bar{\ell}) \to \bar{k}) = gatecount(C) + 1,$$
$$width(id_Q) = |Q|, \qquad width(C; g(\bar{\ell}) \to \bar{k}) = width(C) + \max(0, |\bar{k}| - |\bar{\ell}| - reusable(C)),$$

are both global metrics. $reusable(C)$ is the number of wires reusable from C, obtained by subtracting the number of its outputs from its width.

Example 4.3 (Local Metrics). The circuit metric **Depth** $= ((\mathscr{L} \to \mathbb{N}) \to (\mathscr{L} \to \mathbb{N}), depth)$, where

$$depth(id_Q)(in)(t) = in(t),$$
$$depth(C; g(\bar{\ell}) \to \bar{k})(in)(t) = \begin{cases} \max\{depth(C)(in)(\ell) \mid \ell \in FL(\bar{\ell})\} + 1 & \text{if } t \in FL(\bar{k}), \\ depth(C)(in)(t) & \text{otherwise,} \end{cases}$$

is a local metric. $FL(\bar{\ell})$ is the set of label names occurring in $\bar{\ell}$.

4.3 Resource Aware Circuit Signatures

The notions of global and local metrics that we have just given are able to effectively capture several common circuit resource metrics. Our end goal is to reason about these quantities within a type system, so the next step is deciding how to encode these definitions within a typing judgment. We proceed gradually, starting with a type system for circuits and moving up to circuit description languages only in the next section. Circuits are traditionally amenable to a form of *signature*, which tells us what are the inputs and outputs of a circuit. In the case of CRL, these signatures are derived via a rudimentary typing system which concludes judgments of the form $C : Q \to L$, where Q and L are collections of typed wires. Including global resource information into these judgments is not hard: it suffices to decorate them with a quantity corresponding to the value of the size of C. This leads to a judgment of the form $C : Q \to L; I$, where I is what we previously called an *index*, that is, an arithmetical expression representing a natural number.

Capturing local resources, on the other hand, is not as simple. This is because a local resource metric is not a natural number, but rather a function (and a higher-order function at that). We need our judgments to encode a function from assignments of *something* to input labels to assignments

of *something* to output labels. Now, recall that we said that Q and L are "collections of typed wires". More formally, these are called *label contexts*, and they are defined as partial mappings from label names to wire types. In Section 2.2.2 we informally discussed the possibility of keeping track of local metrics by refining wire types, so at this point it feels natural to rely on Q and L to encode the local metric associated to C. On one hand, we should not make assumptions about the assignment of local resource values to the *input* labels of C, so we decorate the wire types in the codomain of Q with distinct *index variables*. Let Ξ be the set of such variables: it is easy to see that by ranging over all the possible assignments of natural numbers to the variables in Ξ, Q conceptually ranges over all the possible functions from the set of input labels to $\mathbb{N}$. On the other hand, the assignment of local resources to the *output* labels of C should *depend* on the resource values of the input labels, so we decorate the wire types in the codomain of L with index expressions built from the variables in Ξ. The precise form of these expressions depends of course on the shape of the circuit. This way, we can effectively encode a function $(\mathscr{L} \to \mathbb{N}) \to (\mathscr{L} \to \mathbb{N})$ within a judgment, which now has the following form:

$$\Xi \vdash^{s} C : Q \to L; I. \tag{2}$$

This is a *resource-aware circuit signature* (RACS), which tells us that using the variables in Ξ to encode local metrics, circuit C has input Q, output L, and size I. We call Ξ a *local metric context*.

Before we move on the the derivation rules for these judgments, it is worth discussing exactly *what kind of expressions* we use to annotate judgments and wire types. In previous sections, we said that indices are fundamentally arithmetic expressions, but is it enough to allow just the standard arithmetic operators within their language? On one hand, we saw in Section 2 that a satisfactory resource analysis can be carried out using comparatively simple operations, such as maxima and sums. On the other hand, we also saw that different metrics require different operations to happen in different places. If we limited the language of indices to standard arithmetic operations, then we would have to provide *distinct deduction rules* for each metric, even at the basic level of the individual circuit. Because this is unwieldy, we follow a different path. Alongside the standard mathematical operations, we also allow a number of *abstract* resource operators to occur within indices. These operators represent how the local and global metrics evolve as we build up a CRL circuit, but they have no standard arithmetic interpretation by themselves. Rather, they act as placeholders, and are given different meanings depending on the metric under analysis. How many such operators do we need? Ideally, we would like to be able to replicate the recursive definition of circuit metrics as closely as possible. For this, we would need an operator for each CRL constructor and for each kind of metric (global and local). Fortunately, the local metric associated to the identity circuit is always the identity function, so we can make do with just three operators:

- size_W, which represents the global metric associated to the multiset of wire types W. It is used to represent the size of an identity circuit, and the choice of indexing this operator on a multiset tells us that the order of the wires does not matter in this regard.
- $\mathsf{append}_g(I, J, E, F)$, indexed on $g \in \mathscr{G}$, which represents the global metric associated to the application of g to a circuit of size I. Index J represents the size of the wires passing alongside g, while E and F are the size of the wires that go into and come out of g, respectively.
- $\mathsf{out}_{g,n}(I_1, \ldots, I_m)$, indexed on $g \in \mathscr{G}$ and $n \in \mathbb{N}$, which represents the local metric value of the n-th output of g as a function of the local metric values of its m inputs.

This way, we can define a single set of syntactical rules for the derivation of RACSs, which is the one given in Figure 6. These rules in turn represent a *family* of formalisms, indexed on the semantic interpretation of the abstract resource operators, which we write as $\mathfrak{c}$.

We will formally define what $\mathfrak{c}$ is in Section 4.4. For now, let us examine Figure 6. First of all, note that we write "Q, L" as shorthand for $Q \uplus L$, i.e. the union of two contexts with disjoint domains.

$$\textsc{id}\ \frac{\Xi \vdash Q \qquad \vDash_{\mathfrak{c}} \mathsf{size}_{\widetilde{Q}} = I}{\Xi \vdash^{s}_{\mathfrak{c}} id_Q : Q \to Q; I}$$

$$\textsc{append}\ \frac{\Xi \vdash^{s}_{\mathfrak{c}} C : Q \to L, H; I \qquad \mathrm{GTS}_g = (\Delta, T, U) \qquad \delta \in \mathfrak{I}_{\Xi}[\Delta] \qquad \Xi. H \vdash^{b}_{\mathfrak{c}} \bar{\ell} : \delta(T) \qquad \Xi. K \vdash^{b}_{\mathfrak{c}} \bar{k} : \delta(U) \qquad \vDash_{\mathfrak{c}} \mathsf{append}_g(I, \mathsf{size}_{\widetilde{L}}, \mathsf{size}_{\widetilde{H}}, \mathsf{size}_{\widetilde{K}}) = J}{\Xi \vdash^{s}_{\mathfrak{c}} C; g(\bar{\ell}) \to \bar{k} : Q \to L, K; J}$$

$$\textsc{unit}\ \Xi. \bullet \vdash^{b}_{\mathfrak{c}} * : \mathbb{1}$$

$$\textsc{label}\ \frac{\Xi \vDash_{\mathfrak{c}} I = J}{\Xi. \ell : w^I \vdash^{b}_{\mathfrak{c}} \ell : w^J}$$

$$\textsc{tuple}\ \frac{\Xi. Q_1 \vdash^{b}_{\mathfrak{c}} \bar{\ell} : T \qquad \Xi. Q_2 \vdash^{b}_{\mathfrak{c}} \bar{k} : U}{\Xi. Q_1, Q_2 \vdash^{b}_{\mathfrak{c}} \langle \bar{\ell}, \bar{k} \rangle : T \otimes U}$$

Fig. 6. Rules for RACSs and bundle judgments

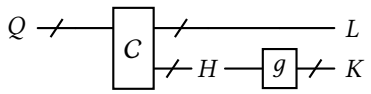

Fig. 7. Routing of wires during the append of an operation g to circuit C

$\Xi. Q \vdash^{b}_{\mathfrak{c}} \bar{\ell} : T$ is what we call a *bundle judgment*, and it tells us that wire bundle $\bar{\ell}$ has type T under label context Q and local metric context Ξ. A label ℓ has type w^I when the local resource value of ℓ is the function of the variables in Ξ described by I. On the other hand, $\Xi \vDash_{\mathfrak{c}} I = J$ is what we call a *semantic judgment*, stating that for all assignments of natural numbers to the variables in Ξ, indices I and J are equal. Moving on to the the actual derivation of RACSs, the ID rule is pretty straightforward: as long as all of the index variables occurring in Q are in Ξ, id_Q has signature $Q \to Q$, which unsurprisingly encodes a local metric corresponding to the identity function. On the side of global metrics, the size of id_Q is equal to $\mathsf{size}_{\widetilde{Q}}$, where $\widetilde{Q}$ denotes the multiset of unannotated wire types occurring in the codomain of Q. Moving on, we find that the APPEND rule is slightly more involved, as it describes the append of a gate g to a circuit C, as shown in Figure 7. Naturally, we need to check that g is appended to wires of the right type. To this effect, we focus on GTS_g, which is the *general type scheme* of g. It is defined as a triple (Δ, T, U), where

- $\Delta = \{i_1, \ldots, i_n\}$ is a set of index variables,
- $T = w_1^{i_1} \otimes w_2^{i_2} \otimes \cdots \otimes w_n^{i_n}$ is the input type of g,
- $U = r_1^{\mathrm{out}_{g,1}(i_1,i_2,\ldots,i_n)} \otimes r_2^{\mathrm{out}_{g,2}(i_1,i_2,\ldots,i_n)} \otimes \cdots \otimes r_m^{\mathrm{out}_{g,m}(i_1,i_2,\ldots,i_n)}$ is the output type of g,

and $w_k, r_k \in \{\mathrm{Bit}, \mathrm{Qubit}\}$, for all k. The role of a type scheme is twofold: on one hand, GTS_g defines the base input and output types of the elementary operation g, and it allows us to type-check its application. On the other hand, these input and output types are annotated with index variables and abstract resource operators, so as to encode the local metric associated to g. In other words, GTS_g provides the same kind information that we get out of RACSs, but at the level of the individual operations in $\mathscr{G}$. A type scheme is instantiated to the appropriate wire annotations via an *index substitution* δ. Informally, $\delta \in \mathfrak{I}_{\Xi}[\Delta]$ means that δ substitutes each of the index variables in Δ with an index whose free variables are in Ξ. On the side of global resources, we have that the global metric value of $C; g(\bar{\ell}) \to \bar{k}$ is given by $\mathsf{append}_g(I, \mathsf{size}_{\widetilde{L}}, \mathsf{size}_{\widetilde{H}}, \mathsf{size}_{\widetilde{K}})$. This is not a surprise, since by the definition of append_g this term describes precisely the size of the circuit shown in Figure 7.

4.4 Circuit Metric Interpretations

As we said earlier, the family of formalisms described by the rules in Figure 6 is indexed on the semantic interpretation of the abstract resource operators, that is, on $\mathfrak{c}$. Formally, $\mathfrak{c}$ is what we call a *circuit metric interpretation* (CMI). A CMI is nothing more that a triple of functions $(\mathfrak{size}_{\mathfrak{c}}^{W}, \mathfrak{append}_{\mathfrak{c}}^{g}, \mathfrak{out}_{\mathfrak{c}}^{g,n})$, where $\mathfrak{size}_{\mathfrak{c}}^{W} \in \mathbb{N}$ interprets size_W, $\mathfrak{append}_{\mathfrak{c}}^{g} : \mathbb{N}^4 \to \mathbb{N}$ interprets append_g, and $\mathfrak{out}_{\mathfrak{c}}^{g,n} : \mathbb{N}^{ar(g)} \to \mathbb{N}$ interprets $\mathrm{out}_{g,n}$. In order to better understand how a CMI can capture a recursive circuit metric, we provide the following example CMIs for the familiar metrics of gate count, width and depth.

*Example 4.4 (***GCount** *CMI).* For gate count, we have $\mathfrak{size}_{\mathbf{GCount}}^{W} = 0$ and $\mathfrak{append}_{\mathbf{GCount}}^{g}(n, l, h, k) = n + 1$. In this case, $\mathfrak{out}_{\mathbf{GCount}}^{g,n}$ is irrelevant and can be chosen to be any constant function. Note that this interpretation exactly encodes the recursive definition of gate count given earlier.

*Example 4.5 (***Width** *CMI).* For width, we have $\mathfrak{size}_{\mathbf{Width}}^{W} = |W|$ and $\mathfrak{append}_{\mathbf{Width}}^{g}(n, l, h, k) = n + \max(0, k - h - (n - (l + h)))$, while $\mathfrak{out}_{\mathbf{Width}}^{g,n}$ is once again irrelevant. Note that this interpretation also encodes the recursive definition of width given earlier, since $k - h$ is exactly $|\bar{k}| - |\bar{\ell}|$ and $n - (l + h)$ is equal to the width of the circuit minus the number of its outputs, i.e. to $\mathit{reusable}(C)$.

*Example 4.6 (***Depth** *CMI).* For depth, we have $\mathfrak{out}_{\mathbf{Depth}}^{g,i}(n_1, \ldots, n_k) = \max(n_1, \ldots, n_k) + 1$, for all g and $1 \leq i \leq \mathrm{inputs}(g)$, while $\mathfrak{size}_{\mathbf{Depth}}^{W}$ and $\mathfrak{append}_{\mathbf{Depth}}^{g}$ are irrelevant and can be chosen to be any constant function. Although it is not as easy to check, we can see that this interpretation encodes the recursive definition of depth given earlier: for the wires that the g is actually appended on, we get a depth of $\max(n_1, \ldots, n_k) + 1$, which is equivalent to $\max\{\mathit{depth}(C)(\mathit{in})(\ell) \mid \ell \in \mathit{FL}(\bar{\ell})\} + 1$, whereas the depth of the other wires is left unchanged, i.e. stays at $\mathit{depth}(C)(\mathit{in})(t)$.

4.5 Soundness of CMIs

In the last three examples, we informally argued that the CMIs we gave capture the recursive definitions of the metrics they are associated with. We now give a more general definition of what it means for a CMI to be *sound* with respect to a given recursive metric. The case of global metrics is straightforward: a CMI $\mathfrak{c}$ is sound with respect to a global metric $\mathscr{R} = (\mathbb{N}, \mu_{\mathscr{R}})$ when the derivability of a judgment of the form $\Xi \vdash_{\mathfrak{c}}^{s} C : Q \to L; I$ entails $\vDash_{\mathfrak{c}} \mu_{\mathscr{R}}(C) \leq I$. Note that we do not require a CMI to be *strictly equal* to the underlying recursive metric. At this stage, we are content with a sound overapproximation. The case of local CMIs is of course more complicated. Recall that, given a local RCM $\mathscr{R} = ((\mathscr{L} \to \mathbb{N}) \to (\mathscr{L} \to \mathbb{N}), \mu_{\mathscr{R}})$, we encode $\mu_{\mathscr{R}}$ in a judgment $\Xi \vdash_{\mathfrak{c}}^{s} C : Q \to L; I$ by encoding the input of $\mu_{\mathscr{R}}$ in Q and its output in L. To do so, we annotate the wire types in Q with the index variables in Ξ and the wire types in L with index expressions that depend on the same variables. This leads to the following definition:

Definition 4.7 (Locally Sound CMI). A CMI $\mathfrak{c}$ is *sound with respect to a local metric* $\mathscr{R} = ((\mathscr{L} \to \mathbb{N}) \to (\mathscr{L} \to \mathbb{N}), \mu_{\mathscr{R}})$ when $\Xi \vdash_{\mathfrak{c}}^{s} C : (\ell_1 : w_1^{i_1}, \ldots, \ell_n : w_n^{i_n}) \to (k_1 : r_1^{I_1}, \ldots, k_m : r_m^{I_m}); J$ entails that for all $j \in \{1, \ldots, m\}$, we have $\Xi \vDash_{\mathfrak{c}} \mu_{\mathscr{R}}(C)(f)(k_j) \leq I_j$, where $f(\ell_h) = i_p$ for $h \in \{1, \ldots, n\}$.

Because we have designed CMIs as to closely mimic the recursive definitions of circuit metrics, and because the formalism of Figure 6 is simple enough, it is not hard to prove that a given CMI is sound with respect to the corresponding recursive metric. In fact, we prove that this is the case for the aforementioned examples:

$$
\begin{array}{llll}
\text{Types} & \mathit{TYPE} & A, B & ::= \mathbb{1} \mid w^I \mid {!^I}\, A \mid A \otimes B \mid A \multimap^I_T B \mid \mathsf{List}_{i<I}\, A \mid \mathsf{Circ}^I_\Theta(T, U) \\
\text{Param. types} & \mathit{PTYPE} & P, R & ::= \mathbb{1} \mid {!^I}\, A \mid P \otimes R \mid \mathsf{List}_{i<I}\, P \mid \mathsf{Circ}^I_\Theta(T, U) \\
\text{Bundle types} & \mathit{BTYPE} & T, U & ::= \mathbb{1} \mid w^I \mid T \otimes U \mid \mathsf{List}_{i<I}\, T \\
\\
\text{Terms} & \mathit{TERM} & M, N & ::= V\ W \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\, V \mid \mathsf{box}_{\Theta,T}\, V \\
 & & & \mid \mathsf{apply}(V, W) \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \mathsf{fold}_i\ V\ W\ X \\
\text{Values} & \mathit{VAL} & V, W & ::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\bar{\ell}, C, \bar{k}) \mid \langle V, W\rangle \\
 & & & \mid \mathsf{nil} \mid \mathsf{rcons}\ V\ W \\
\text{Wire bundles} & \mathit{BVAL} & \bar{\ell}, \bar{k} & ::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{rcons}\ \bar{\ell}\ \bar{k} \\
\text{Indices} & \mathit{INDEX} & I, J & ::= n \mid i \mid I + J \mid I - J \mid I \cdot J \mid \max(I, J) \\
 & & & \mid \mathsf{size}_W \mid \mathsf{append}_g(I, J, E, F) \mid \mathsf{out}_{g,n}(I_1, \ldots, I_m) \\
 & & & \mid \mathsf{e} \mid \mathsf{wire}_w \mid I \circledast J \mid I \oplus J \mid \circledast_{i<I} J \mid \bigoplus_{i<I} J
\end{array}
$$

Fig. 8. Types and syntax of Proto-Quipper-RA

Proposition 4.8. *The* **GCount**, **Width** *and* **Depth** *CMIs given in examples 4.4, 4.5 and 4.6 are sound with respect to the corresponding recursive metrics given in examples 4.2 and 4.3.*

Proof. By induction on the derivation of $\Xi \vdash^s_{\mathfrak{c}} C : Q \to L; I$, for $\mathfrak{c} \in \{\mathbf{GCount}, \mathbf{Width}, \mathbf{Depth}\}$. □

5 From Circuits to Circuit Description Languages: Resource-Aware Proto-Quipper

In this section we take the step from the low-level CRL to an actual circuit description language. We present *Resource Aware Proto-Quipper*, or Proto-Quipper-RA for short: an extension of the Proto-Quipper language introduced in Section 3 equipped with a family of type-and-effect systems that supports generic reasoning about resources. Unsurprisingly, this step up in expressiveness introduces a number of new challenges with respect to the previous section. This is because although the object of analysis remains the same (quantum circuits), the structure of programs is fundamentally richer than those of circuits, allowing for greater parametricity and compositionality on one side, but requiring more sophisticated abstractions to keep track of resources on the other.

5.1 Types and Terms

The types and syntax of Proto-Quipper-RA are given in Figure 8. On the side of types, we can see that the standard Proto-Quipper types have been *refined* with quantitative information useful for resource analysis. Let us take a moment to dissect these refinements. Naturally, we inherit from circuit signatures the annotated wire type w^I, which encodes information about the local metric value of the corresponding label in I. Speaking of circuits, the boxed circuit type $\mathsf{Circ}^I_\Theta(T, U)$ is annotated with I, which carries information about the global metric associated to the circuit. Local metric information is instead encoded in T and U, using locally-scoped index variables taken from Θ. The linear arrow type $A \multimap^I_T B$ is annotated with index I and bundle type T. Like we mentioned in Section 2, I represents the global metric associated with the circuit built by the function once we apply it to a value of type A, while T is the type of the wires enclosed in the function's closure. The annotation on ${!^I}\, A$ is similar to the arrow's and tells us that resuming (forcing) a suspended (lifted) computation of that type has the side effect of building a circuit of size at most I. Lastly, the list type also gets annotated with quantitative information: the type $\mathsf{List}_{i<I}\, A$ represents lists of exactly I elements, where the i-th element has type A which can depend on i. We call this type a *sized dependent list type*. Although this type might seem cumbersome to work with, size and dependency are essential for resource analysis: if we did not have size, then we would have no bound on how

many wires are potentially in a list, whereas if we did not have dependency we would only be able to work with lists of wires with *homogeneous* local metrics. Furthermore, the fact that the size is *exact* allows us to treat lists of wires as wire bundles and feed them into circuits, since they are now functionally equivalent to finite tensors of wires.

On the side of the language of terms, we have but minor changes from the Proto-Quipper definition of Section 3: $\mathsf{box}_{\Theta,T}\ V$ now declares the set of local variables Θ used to encode local metrics in T, and the fold operator binds an index variable i. This variable appears in the type of the step function and it allows each step of the fold to contribute *differently* to the size of the constructed circuit. Lastly, note that lists now grow towards the right, with $\mathsf{rcons}\ V\ W$ appending W at the *end* of list V. This allows to work more smoothly with the new dependent aspect of lists.

Index Terms. In moving up to a higher level language, we keep the same approach to indices that we established in Section 4, equipping their language with abstract operators that are interpreted differently depending on the resource being analyzed. The final language of indices, given in Figure 8, comprises therefore natural numbers, index variables and standard arithmetic operators, the cicuit-level abstract operators introduced in Section 4 and lastly a set of new abstract operators and constants that relate to the particular way that circuits are built in Proto-Quipper. In particular, individual circuits are no longer the focus of the resource analysis, but rather *computations*. Annotations at the level of Proto-Quipper-RA therefore describe the size associated with the circuits produced by these computations. The new operators include:

- e, which represents the concept of null size,
- wire_w, which describes the size of a single wire of type w,
- $I \circledast J$, which describes the size of the circuit built by the execution *in sequence* of two computations that build circuits of size I and J, respectively,
- $I \oplus J$, which describes the size of the circuit built by the execution *in parallel* of two computations that build circuits of size I and J, respectively.

Note that $\circledast$ and $\oplus$ also have their bounded versions: $\circledast_{i<I} J$ and $\bigoplus_{i<I} J$, which describe the composition in sequence (respectivey, in parallel) of a family of I computations, each annotated with global metric J, where J may depend on i. The attentive reader might have noticed that all of these operators talk exclusively about *size*, that is, *global* metrics. This is because moving from CRL to Proto-Quipper-RA (i.e. from a low-level, gate-by-gate circuit description approach to a more compositional one) has an impact on how we handle *global* metrics, but it does not change the way that we analysize *local* metrics: the way that wires are routed around in CRL and Proto-Quipper-RA is fundamentally the same, and as such Proto-Quipper-RA inherits the ability of performing local metric analysis directly from CRL's RACSs, for free. This is why we do not need to introduce new index operators to deal with local metrics.

5.2 Typing Rules

The idea of a type system in which computations are annotated with information regarding the side effect they produce (in this case, the size of the circuit they build) is not new and takes the name of a *type-and-effect system* [53]. In the case of Proto-Quipper-RA, this means that a typing judgments on a term M gets annotated with an index I, denoting an upper bound to the size of the circuits produced by M. We say *circuits* because M can describe a *family* of circuits, depending on some natural number parameters encoded as index variables. We are therefore dealing with a computational typing judgment of the form

$$\Theta; \Xi.\, \Gamma; Q \vdash^{c} M : A; I, \tag{3}$$

which reads as "for all assignments of natural numbers to the index variables in Θ and Ξ, under typing context Γ and label context Q, term M has type A and produces a circuit of size at most I". Here, Γ is a traditional linear/nonlinear typing context, while Q is a label context like the ones we introduced in Section 4. If Γ only contains parameter variables, we write it as Φ. Note that we employ two different, disjoint index variable contexts, Θ and Ξ. Why is that? That is because at the level of Proto-Quipper-RA, index variables play two fundamentally distinct roles. One one hand, the variables in Θ represent the natural number parameters that a computation may depend on, and they effectively allow M to describe a *family* of circuits, as opposed to an individual circuit. The variables in Θ are the *only* parameters that types and effects are allowed to depend on, which is why we separate them from the other parameters in Γ in the first place. This allows us to restrict dependency to natural-valued parameters, sacrificing a little bit of expressiveness in favor of automation (see Section 7). On the other hand, the index variables in Ξ are only used to capture local metrics, in a way similar to RACSs. We assume the existence of an unspecified underlying circuit C, such that Q represents the outputs of C that are manipulated by M and Ξ represents the variables used to encode the local metrics of C. Because of this, we have that the wire type annotations in Q depend exclusively on the variables in Ξ, while the wire type annotations in A, which intuitively correspond to the local metrics associated with the outputs of the circuit built by M, may also depend on the parameters in Θ. Note that while the variables in Θ have operational value, those in Ξ do not, as they are only used to syntactically encode a local metric function.

Besides the main computational judgment, we also work with a judgment for values of the form $\Theta; \Xi. \Gamma; Q \vdash^{v} V : A$. The latter is identical to the former, save for the absence of an effect annotation, since values are effect-less. That being said, the main typing rules for deriving these two kinds of judgment are given in Figure 9, where $A\{I/i\}$ denotes the capture-avoiding substitution of index I for variable i in type A, $\Theta \vdash I$ means that all of the index variables occurring in I are in Θ, and $\Theta; \Xi \vdash A$ means that all of the index variables mentioned in type A are in Θ, with the exception of those occurring in the annotations of wire types, which may also come from Ξ. Let us examine the rules in Figure 9 in more detail, focusing on their metric estimation logic.

5.2.1 Typing Circuit Applications. We start with the fundamental APPLY rule. Value V has type $\mathsf{Circ}^{I}_{\Delta}(T, U)$, which means that it is a boxed circuit of size I whose local metrics are encoded in T and U using the variables in Δ. Therefore, if we manage to find a $\delta \in \mathfrak{I}_{\Theta,\Xi}[\Delta]$ that substitutes the variables in Δ in such a way that $\delta(T)$ matches the type of the application argument W, then we can safely apply V to the underlying circuit, obtaining output wires of type $\delta(U)$. The size upper bound is clearly that of the applied circuit, i.e. I. As for local metrics, it comes as no surprise that the APPLY rule is reminiscent of the APPEND rule from Figure 6: the type $\mathsf{Circ}^{I}_{\Delta}(T, U)$ effectively acts as a general type scheme (Δ, T, U), and δ instantiates the local index variables in Δ with the actual variable names used to encode metrics in the current type derivation. At this point, one might wonder: where do we get the metric information associated to a boxed circuit? Looking at the CIRC rule, we can see that all of the information that we can obtain through RACSs can also be reflected in Proto-Quipper-RA's type system in the form of a circuit type: if C has input Q, output L and size I, and Q and L confer types T and U to $\bar{\ell}$ and $\bar{k}$, respectively, then the boxed circuit $(\bar{\ell}, C, \bar{k})$ has type $\mathsf{Circ}^{J}_{\Delta}(T, U)$, where $J = \max(\#(Q), I, \#(L))$ overapproximates the size of the circuit as the maximum between I and the size of either Q or L, for technical reasons. Note that the judgments are parametrized by a CMI $\mathfrak{c}$, which is used to obtain this information and therefore serves as a sort of "ground truth" regarding the size and local metrics of the elementary gates and operations encoded as boxed circuits in Proto-Quipper-RA.

$$\textsc{var}\ \frac{}{\Theta; \Xi \vdash \Phi, x : A}{\Theta; \Xi. \Phi, x : A; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} x : A} \qquad \textsc{lab}\ \frac{\Theta; \Xi \vdash \Phi \qquad \Xi \vdash I}{\Theta; \Xi. \Phi; \ell : w^I \vdash^{v}_{\mathsf{t},\mathfrak{c}} \ell : w^I}$$

$$\textsc{abs}\ \frac{\Theta; \Xi. \Gamma, x : A; Q \vdash^{c}_{\mathsf{t},\mathfrak{c}} M : B; I \qquad T = wc(\Gamma; Q; M)}{\Theta; \Xi. \Gamma; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} \lambda x_A.M : A \multimap^{I}_{T} B} \qquad \textsc{app}\ \frac{\Theta; \Xi. \Phi, \Gamma_1; Q_1 \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : A \multimap^{I}_{-} B \qquad \Theta; \Xi. \Phi, \Gamma_2; Q_2 \vdash^{v}_{\mathsf{t},\mathfrak{c}} W : A}{\Theta; \Xi. \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash^{c}_{\mathsf{t},\mathfrak{c}} V\ W : B; I}$$

$$\textsc{circ}\ \frac{\Delta \vdash^{s}_{\mathfrak{c}} C : Q \to L; I \qquad J = \max(\#(Q), I, \#(L)) \qquad \emptyset; \Delta. \bullet; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} \bar{\ell} : T \qquad \emptyset; \Delta. \bullet; L \vdash^{v}_{\mathsf{t},\mathfrak{c}} \bar{k} : U \qquad \Theta; \Xi \vdash \Phi}{\Theta; \Xi. \Phi; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} (\bar{\ell}, C, \bar{k}) : \mathsf{Circ}^{J}_{\Delta}(T, U)}$$

$$\textsc{box}\ \frac{E = (I \oplus \#(T)) \circledast J \qquad \Theta; \Delta. \Phi; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : !^{I}(T \multimap^{J}_{-} U)}{\Theta; \Xi. \Phi; \bullet \vdash^{c}_{\mathsf{t},\mathfrak{c}} \mathsf{box}_{\Delta,T}\ V : \mathsf{Circ}^{E}_{\Delta}(T, U); \mathsf{e}} \qquad \textsc{apply}\ \frac{\Theta; \Xi. \Phi, ; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : \mathsf{Circ}^{I}_{\Delta}(T, U) \qquad \delta \in \mathfrak{F}_{\Theta,\Xi}[\Delta] \qquad \Theta; \Xi. \Phi, \Gamma; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} W : \delta(T)}{\Theta; \Xi. \Phi, \Gamma; Q \vdash^{c}_{\mathsf{t},\mathfrak{c}} \mathsf{apply}(V, W) : \delta(U); I}$$

$$\textsc{return}\ \frac{\Theta; \Xi. \Gamma; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : A}{\Theta; \Xi. \Gamma; Q \vdash^{c}_{\mathsf{t},\mathfrak{c}} \mathsf{return}\ V : A; \#(A)} \qquad \textsc{let}\ \frac{E = (I \oplus \#(\Gamma_2, Q_2)) \circledast J \qquad \Theta; \Xi. \Phi, \Gamma_1; Q_1 \vdash^{c}_{\mathsf{t},\mathfrak{c}} M : A; I \qquad \Theta; \Xi. \Phi, \Gamma_2, x : A; Q_2 \vdash^{c}_{\mathsf{t},\mathfrak{c}} N : B; J}{\Theta; \Xi. \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash^{c}_{\mathsf{t},\mathfrak{c}} \mathsf{let}\ x = M\ \mathsf{in}\ N : B; E}$$

$$\textsc{nil}\ \frac{\Theta; \Xi \vdash \Phi \qquad \Theta, i; \Xi \vdash A}{\Theta; \Xi. \Phi; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} \mathsf{nil} : \mathsf{List}_{i<0} A} \qquad \textsc{rcons}\ \frac{\Theta; \Xi. \Phi, \Gamma_1; Q_1 \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : \mathsf{List}_{i<I} A \qquad \Theta; \Xi. \Phi, \Gamma_2; Q_2 \vdash^{v}_{\mathsf{t},\mathfrak{c}} W : A\{I/i\}}{\Theta; \Xi. \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash^{v}_{\mathsf{t},\mathfrak{c}} \mathsf{rcons}\ V\ W : \mathsf{List}_{i<I+1} A}$$

$$\textsc{fold}\ \frac{\begin{array}{c}\Theta, i; \Xi. \Phi; \bullet \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : !^{I}((B \otimes A\{E - (i+1)/j\}) \multimap^{J}_{-} B\{i+1/i\}) \\ \Theta; \Xi. \Phi, \Gamma_1; Q_1 \vdash^{v}_{\mathsf{t},\mathfrak{c}} W : B\{0/i\} \qquad \Theta; \Xi. \Phi, \Gamma_2; Q_2 \vdash^{v}_{\mathsf{t},\mathfrak{c}} X : \mathsf{List}_{j<E} A \\ F = \circledast_{i<E}(((I \oplus \#(B \otimes A\{E - (i+1)/j\})) \circledast J) \oplus \bigoplus_{j<E-(i+1)} \#(A))\end{array}}{\Theta; \Xi. \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash^{c}_{\mathsf{t},\mathfrak{c}} \mathsf{fold}_i\ V\ W\ X : B\{E/i\}; \max(\#(B\{0/i\}), F)}$$

$$\textsc{vsub}\ \frac{\Theta; \Xi. \Gamma; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : A \qquad \Theta; \Xi \vdash_{\mathsf{t},\mathfrak{c}} A <: B}{\Theta; \Xi. \Gamma; Q \vdash^{v}_{\mathsf{t},\mathfrak{c}} V : B} \qquad \textsc{csub}\ \frac{\Theta; \Xi. \Gamma; Q \vdash^{c}_{\mathsf{t},\mathfrak{c}} M : A; I \qquad \Theta; \Xi \vdash_{\mathsf{t},\mathfrak{c}} A <: B \qquad \Theta \vDash_{\mathsf{t}} I \leq J}{\Theta; \Xi. \Gamma; Q \vdash^{c}_{\mathsf{t},\mathfrak{c}} M : B; J}$$

Fig. 9. Main typing rules of Proto-Quipper-RA

5.2.2 LET, BOX and the Size of Types. Previously, we indirectly referred to $\#(Q)$ as the "size" of Q. The CIRC rule is by no means the only place where the $\#(\cdot)$ function is used, so let us take a moment to define it properly. Recall that wires have a size of their own, which at this level of abstraction

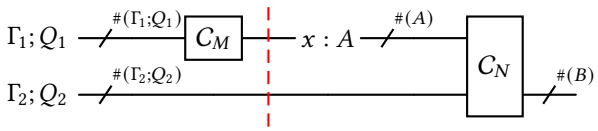

Fig. 10. The shape of a circuit built by a let expression. The dashed line separates the scopes of M and N.

is represented by the abstract indices wire_w. Values may contain wires, which means that even though they do not *actively* build circuits, they still have a size associated with them, corresponding to the aggregated size of their wires. This size has obviously an impact on global resource analysis, which is why it is imperative that we keep track of it in our type system. Fortunately, we find that the size of a value can be computed as a function of its type: we call $\#(\cdot) : \mathit{TYPE} \to \mathit{INDEX}$ the *type size function*, and we call $\#(A)$ the *size of* A, which we define as follows:

$$\#(\mathbb{1}) = \#(!^I A) = \#(\mathsf{Circ}^I_\Theta(T,U)) = \mathsf{e}, \qquad \#(w^I) = \mathsf{wire}_w, \qquad \#(A \otimes B) = \#(A) \oplus \#(B),$$

$$\#(\mathsf{List}_{i<I} A) = \bigoplus_{i<I} \#(A), \qquad \#(A \multimap^I_T B) = \#(T).$$

Note how the case for the arrow type fundamentally depends on its bundle type annotation T: if we were working with regular arrow types, we would not be able to know which wires from the environment are captured by the function, as their types would not be not guaranteed to occur in either the domain or the codomain of the arrow. This is the main reason why we need the T annotation in the first place. Typing and label contexts can also contain wires, so we lift the definition of $\#(\cdot)$ to contexts in the natural way: we write $\#(Q)$ to represent the combined size of the wires occurring in Q, and we write $\#(\Gamma; Q)$ as shorthand for $\#(\Gamma) \oplus \#(Q)$.

The LET rule is without a doubt the rule in which the type size function plays its most crucial role: before they are consumed by N, the wires in Γ_2 and Q_2 must travel alongside the circuit of size I that is built by M, and as such their size is taken into account in the upper bound for the left-hand side of the let, which becomes $I \oplus \#(\Gamma_2; Q_2)$. Term N then builds a circuit of size at most J, leading to an overall upper bound of $(I \oplus \#(\Gamma_2; Q_2)) \circledast J$. This reasoning is better illustrated in Figure 10. We encounter a similar scenario in the BOX rule: the semantics of a term like $\mathsf{box}_{\Delta,T}\, V$ involves the forcing of function V, followed by its application to a dummy argument of type T (see Figure 12). The forcing of V produces a circuit of size I, while the argument of size $\#(T)$ travels alongside. Then, the execution of the function produces a circuit of size J, leading to an overall circuit size of $E = (I \oplus \#(T)) \circledast J$.

5.2.3 Typing Folds. Unsurprisingly, the FOLD rule also makes use of the type size function. Although intimidating, the rule can be interpreted as a more general form of the LET rule. Recall that lists now grow to the right, which means that folds consume them from right to left. During iteration i, we force the step function V and apply it to the $(E-(i+1))$-th element of X, which has type $A\{E-(i+1)/j\}$, and to the accumulator, which has type B. The resulting subcircuit has a shape similar to the one analyzed in the BOX rule, with a size upper bound of $(I \oplus \#(B \otimes A\{E-(i+1)/j\})) \circledast J$. At the same time, the remaining $E-(i+1)$ elements of the list, of collective size $\bigoplus_{j<E-(i+1)} \#(A)$, pass alongside this subcircuit, leading the overall subcircuit built by iteration i to have a size upper bound of $((I \oplus \#(B \otimes A\{E-(i+1)/j\})) \circledast J) \oplus \bigoplus_{j<E-(i+1)} \#(A)$. Sequencing together E such iterations, we obtain an upper bound that coincides with the one described by index term F.

5.2.4 The Wire Content of Functions. Earlier, we justified the need for the T annotation in the arrow type. But how is T computed, exactly? Looking at the ABS rule, we can see $T = wc(\Gamma; Q; M)$.

$$\textsc{wire}\ \frac{\Theta, \Xi \vDash_{\mathfrak{t},\mathfrak{c}} I \leq J}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} w^I <: w^J} \qquad \textsc{bang}\ \frac{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A <: B \qquad \Theta \vDash_{\mathfrak{t},\mathfrak{c}} I \leq J}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} !^I A <: !^J B}$$

$$\textsc{tensor}\ \frac{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A_1 <: A_2 \qquad \Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} B_1 <: B_2}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A_1 \otimes B_1 <: A_2 \otimes B_2} \qquad \textsc{arrow}\ \frac{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A_2 <: A_1 \qquad \Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} B_1 <: B_2 \qquad \Theta \vDash_{\mathfrak{t},\mathfrak{c}} I_1 \leq I_2 \qquad \Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} T_1 <: T_2}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A_1 \multimap_{T_1}^{I_1} B_1 <: A_2 \multimap_{T_2}^{I_2} B_2}$$

$$\textsc{list}\ \frac{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} A <: B \qquad \Theta \vDash I = J}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} \mathsf{List}_{i<I} A <: \mathsf{List}_{i<J} B} \qquad \textsc{circ}\ \frac{\Theta; \Delta \vdash_{\mathfrak{t},\mathfrak{c}} T_1 <:> T_2 \qquad \Theta; \Delta \vdash_{\mathfrak{t},\mathfrak{c}} U_1 <:> U_2 \qquad \Theta \vDash_{\mathfrak{t},\mathfrak{c}} I \leq J}{\Theta; \Xi \vdash_{\mathfrak{t},\mathfrak{c}} \mathsf{Circ}_\Delta^I(T_1, U_1) <: \mathsf{Circ}_\Delta^J(T_2, U_2)}$$

Fig. 11. The subtyping rules of Proto-Quipper-RA

Here, $wc(\cdot)$ is what we call the *wire content function*, which is originally defined on types. What $wc(A)$ does is "strip" A of anything but wire types, obtaining a bundle type that represents all and only the wires that are contained in a value of type A:

$$wc(\mathbb{1}) = wc(!^I A) = wc(\mathsf{Circ}_\Theta^I(T, U)) = \mathbb{1}, \qquad wc(w^I) = w^I, \qquad wc(A \otimes B) = wc(A) \otimes wc(B),$$

$$wc(\mathsf{List}_{i<I} A) = \mathsf{List}_{i<I} wc(A), \qquad wc(A \multimap_T^I B) = T.$$

The $wc(\cdot)$ function is similar to $\#(\cdot)$, but it preserves more information. In fact, note that $\#(wc(A)) = \#(A)$ for all A. In lifting $wc(\cdot)$ to contexts, instead of assuming a total ordering of label and variable names, we use the structure of M as a guide to determine the shape of the resulting bundle type, in an approach that is vaguely reminiscent of *coeffects* [54]. This may seem like an unnecessarily complicated step, but it is a good way of ensuring that the structure of T is preserved under variable substitution, something that is essential for the correctness proof of Section 6 and for a future denotational account of Proto-Quipper-RA.

5.2.5 Subtyping. We conclude the exposition of the typing rules with a brief account of subtyping in Proto-Quipper-RA. Being based in part on type refinements, our type system unsurprisingly features two subsumption rules: VSUB for values and CSUB for computations. These rules rely on a subtyping judgment of the form $\Theta; \Xi \vdash A <: B$, which informally reads "for all assignments of natural numbers to the index variables in Θ and Ξ, A is at least as refined as B". In the context of resources analysis, this usually means that A describes a resource consumption bound no greater than the one described by B. The subtyping rules of Proto-Quipper-RA are given in Figure 11.

5.3 Resource Metric Interpretations

Besides the CMI $\mathfrak{c}$, which provides the ground truth in the matter of the size of circuits, the typing judgments in Figure 9 are also parametrized on a second interpretation $\mathfrak{t}$, which we call a *resource metric interpretation* (RMI) and which specifically interprets the abstract operators introduced with Proto-Quipper-RA. An RMI is therefore a quadruple $(\mathfrak{e}_\mathfrak{t}, \mathfrak{wire}_\mathfrak{t}^w, \mathfrak{seq}_\mathfrak{t}, \mathfrak{par}_\mathfrak{t})$, where $\mathfrak{e}_\mathfrak{t} \in \mathbb{N}$ interprets e, $\mathfrak{wire}_\mathfrak{t}^w \in \mathbb{N}$ interprets wire_w, $\mathfrak{seq}_\mathfrak{t} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ interprets $\circledast$ and its bounded counterpart and $\mathfrak{par}_\mathfrak{t} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ interprets $\oplus$ and its bounded counterpart. Recall when in Section 2 we discussed informally the size of wires and how different global metrics are combined in sequence and parallel. In the following examples, we can finally give a formal account of those intuitions:

Example 5.1 (Gate Count RMI). In the case of gate count we have $\mathfrak{e}_{\mathbf{Gatecount}} = 0$, $\mathfrak{wire}^{w}_{\mathbf{Gatecount}} = 0$ for all w, and $\mathfrak{seq}_{\mathbf{Gatecount}} = \mathfrak{par}_{\mathbf{Gatecount}} = +$.

Example 5.2 (Width RMI). In the case of width we have $\mathfrak{e}_{\mathbf{Width}} = 0$, $\mathfrak{wire}^{w}_{\mathbf{Width}} = 1$ for $w \in \{\mathsf{Qubit}, \mathsf{Bit}\}$, $\mathfrak{seq}_{\mathbf{Width}} = \max$ and $\mathfrak{par}_{\mathbf{Width}} = +$.

At this point, a question arises naturally: should we consider any and all possible interpretations of the aforementioned resource operators? Or can we safely make some assumptions about the properties of the RMIs we work with, as to simplify reasoning? The answer is clearly that we *do* want to impose some constraints on RMIs. We say that an RMI $\mathfrak{t}$ is *well-behaved* when it satisfies the following properties:

(1) $(\mathbb{N}, \mathfrak{par}_{\mathfrak{t}})$ is an ordered commutative monoid with identity $\mathfrak{e}_{\mathfrak{t}}$,
(2) $(\mathbb{N}, \mathfrak{seq}_{\mathfrak{t}})$ is an ordered monoid with identity $\mathfrak{e}_{\mathfrak{t}}$,
(3) $\mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, \mathfrak{seq}_{\mathfrak{t}}(n, m)) = \mathfrak{seq}_{\mathfrak{t}}(\mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, n), \mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, m))$ for all $w \in \mathscr{W}, n, m \in \mathbb{N}$.
(4) Either $\forall w.\mathfrak{wire}^{w}_{\mathfrak{t}} = \mathfrak{e}_{\mathfrak{t}}$ or $\forall n, m.n \leq m \Rightarrow \mathfrak{seq}_{\mathfrak{t}}(n, m) = \mathfrak{seq}_{\mathfrak{t}}(m, n) = m$.

Intuitively, well-behavedness describes what we expect from a "reasonable" resource analysis. In particular, the requirements that the monoids in constraints (1) and (2) be ordered reflects the idea that when we compose a circuit C with another circuit $\mathcal{D}$, the result is *at least as big* as either C or $\mathcal{D}$, i.e. $\mathfrak{seq}_{\mathfrak{t}}$ and $\mathfrak{par}_{\mathfrak{t}}$ should be monotonically non-decreasing. The additional commutativity of $\mathfrak{par}_{\mathfrak{t}}$ reflects the idea that that computations executing in parallel are independent, and thus their relative ordering is irrelevant for resource analysis. Constraint (3) is more subtle, and tells us that $\mathfrak{par}_{\mathfrak{t}}$ distributes over $\mathfrak{seq}_{\mathfrak{t}}$ when wires are involved. Finally, constraint (4) tells us that any metric that takes wires into consideration (e.g. width) should also take into account their reuse during sequential composition. Naturally, the RMIs given in examples 5.1 and 5.2 are easily proven to be well-behaved. In fact, we assume that from now on all the RMIs we deal with are well-behaved.

Note that these constraints are *not* meant to guarantee the correctness of Proto-Quipper-RA (which is the subject of Section 6). Rather, we introduce them to simplify the reasoning about the type system, as they allow us to make some assumptions about the behavior of the RMIs we work with. Note also that we do not require analogous constraints to hold for the much simpler CMIs.

5.4 Operational Semantics

We define the operational semantics of Proto-Quipper-RA by means of a big-step evaluation relation on *configurations*. A configuration consists of a circuit C and a term M that can access C's output wires and thus extend the circuit with new operations. We write $(C, M) \Downarrow (\mathcal{D}, V)$ when M evaluates to V and turns C into $\mathcal{D}$ as a side effect. Operationally speaking, Proto-Quipper-RA does not introduce any novel language features to the Proto-Quipper family. As such, we only discuss the three rules of most interest, which are shown in Figure 12, referring the reader to [8] or even [57] for a more comprehensive treatment of the Proto-Quipper operational semantics.

The APPLY rule describes how apply extends the underlying circuit C with subcircuit $\mathcal{D}$. The concatenation of $\mathcal{D}$ to C is delegated to the *emit* function. This function is given in Definition 5.3, where we write $(\bar{\ell}, C, \bar{k}) \cong (\bar{\imath}, \mathcal{D}, \bar{q})$ to say that two boxed circuits are *equivalent*, i.e. that they are the same circuit up to a renaming of labels. This amounts to a form of α-equivalence for circuits.

Definition 5.3 (emit). We define *the emission of* $(\bar{\ell}, \mathcal{D}, \bar{k})$ *to* C *on* $\bar{\imath}$, written $\mathit{emit}(C, \bar{\imath}, (\bar{\ell}, \mathcal{D}, \bar{k}))$, as a pair of circuit and wire bundle computed as follows:

(1) Find $(\bar{\imath}, \mathcal{D}', \bar{q}) \cong (\bar{\ell}, \mathcal{D}, \bar{k})$ such that the labels shared by C and $\mathcal{D}'$ are exactly those in $\bar{\imath}$,
(2) Return $(C :: \mathcal{D}', \bar{q})$.

On the other hand, the semantics of a term of the form $\mathrm{box}_{\Delta,T}(\mathsf{lift}\, M)$ relies on the *freshlabels* function, which takes as input a bundle type T and instantiate fresh $Q, \bar{\ell}$ such that $\bar{\ell}$ has type T

$$\textsc{apply}\ \frac{(\mathcal{E}, \bar{q}) = \mathit{emit}(C, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))}{(C, \mathsf{apply}((\bar{\ell}, \mathcal{D}, \bar{k}), \bar{t})) \Downarrow (\mathcal{E}, \bar{q})} \qquad \textsc{box}\ \frac{(Q, \bar{\ell}) = \mathit{freshlabels}(T) \quad (\mathit{id}_Q, M) \Downarrow (\mathcal{D}, V) \quad (\mathcal{D}, V\ \bar{\ell}) \Downarrow (\mathcal{E}, \bar{k})}{(C, \mathsf{box}_{\Delta,T}(\mathsf{lift}\, M)) \Downarrow (C, (\bar{\ell}, \mathcal{E}, \bar{k}))}$$

$$\textsc{fold-step}\ \frac{(C, M\{0/i\}) \Downarrow (\mathcal{D}, Y) \quad (\mathcal{D}, Y\ \langle V, X\rangle) \Downarrow (\mathcal{E}, Z) \quad (\mathcal{E}, \mathsf{fold}_i\ (\mathsf{lift}\, M\{i+1/i\})\ Z\ W) \Downarrow (\mathcal{F}, S)}{(C, \mathsf{fold}_i\ (\mathsf{lift}\, M)\ V\ (\mathsf{rcons}\ W\ X)) \Downarrow (\mathcal{F}, S)}$$

Fig. 12. An excerpt of Proto-Quipper-RA's big-step operational semantics

under Q. The wire bundle $\bar{\ell}$ is then passed as an argument to the circuit-building function V, and the resulting computation is evaluated in the context of the identity circuit id_Q. The result is a standalone circuit $\mathcal{E}$, together with its output labels $\bar{k}$. Eventually, $\bar{\ell}$ and $\bar{k}$ become respectively the input and output interfaces of the boxed circuit $(\bar{\ell}, \mathcal{E}, \bar{k})$, which is the result of the evaluation.

Lastly, the FOLD-STEP rule is interesting from the point of view of the operational value of indices. Earlier we said that fold binds a variable name i, which allows each iteration of the fold to contribute differently to the overall circuit being built. This mechanism is evident in this rule: when we run the first iteration, i is instantiated to 0 and the resulting step function is applied to the accumulator. Next, the result of the iteration becomes the new accumulator, we increment i and we recur on the rest of the list. Note that V, W, X, Y, Z and S are all values.

6 Correctness

In this section we define what it means for a Proto-Quipper-RA judgment to be *correct*, and we prove that a Proto-Quipper-RA instance defined by an RMI $\mathfrak{t}$ and a CMI $\mathfrak{c}$ is correct by construction under reasonable assumptions about the relationship between $\mathfrak{t}$ and $\mathfrak{c}$.

6.1 Defining Correctness

Any property of a Proto-Quipper-RA judgment fundamentally depends on the chosen RMI and CMI: the former describes the high-level analysis carried out at the language level, whilst the second describes (an overapproximation of) the actual size of the circuits. For this reason, from now on, when we talk of correctness, we are really talking about the correctness *of* an RMI *with respect* to a CMI. If an RMI gives us an instance of Proto-Quipper-RA that is correct with respect to a CMI, and that CMI soundly overapproximates a ground truth recursive circuit metric, then we can rightfully say that the analyses carried out through the relevant Proto-Quipper-RA instance are correct.

Secondly, we must distinguish between the correctness of the analysis of global and local resources. On the side of global metrics, correctness is fairly straightforward: a judgment $\emptyset; \Xi. \bullet; Q \vdash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$ is correct when, for all C supplying the wires in Q, M evaluates to V, extending C with a subcircuit $\mathcal{D}$ whose size (according to $\mathfrak{c}$) is at most I. On the side of refinement types and local metrics, correctness is a bit more involved: let L be the label context corresponding to the labels output by $\mathcal{D}$. Due to linearity, the labels in the domain of L must appear *somewhere* in V. Similarly, the wire types in the codomain of L, alongside their local metric annotations, must appear *somewhere* in A. Thus, correctness in the sense of local resources is a matter of matching the annotations in L with the annotations in A using V as a bridge, and then checking that the annotations in A soundly overapproximate the corresponding annotations in L. This idea is formalized in the following definition of *local resource correctness*.

Definition 6.1 (Local Resource Correctness). We define $LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q, V, A)$ as the smallest relation such that the following conditions hold:

$$
\begin{aligned}
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(\bullet, V, \mathbb{1}), \qquad LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(\bullet, V, !^I A), \qquad LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(\bullet, V, \mathsf{Circ}^I_\Delta(T, U)),\\
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(\bullet, \mathsf{nil}, \mathsf{List}_{i<I} A) &\Leftrightarrow \vDash I = 0,\\
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(\ell : w^I, \ell, w^J) &\Leftrightarrow \Xi \vDash_{\mathfrak{t},\mathfrak{c}} I \leq J,\\
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q, \lambda x_A.M, A \multimap^I_T B) &\Leftrightarrow LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q, \mathit{labs}(M), T),\\
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q \uplus L, \langle V, W\rangle, A \otimes B) &\Leftrightarrow LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q, V, A) \wedge LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(L, W, B),\\
LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q \uplus L, \mathsf{rcons}\ V\ W, \mathsf{List}_{i<I} A) &\Leftrightarrow \exists J.(\vDash I = J + 1 \wedge LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(Q, V, \mathsf{List}_{i<J} A)\\
&\qquad \wedge LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(L, W, A\{J/i\})).
\end{aligned}
$$

Here $\mathit{labs} : \mathit{TERM} \cup \mathit{VAL} \to \mathit{BVAL}$ is a function that strips a language expression of everything that is not a wire bundle, in the same way that $wc(\cdot)$ strips a type of anything that is not a wire type. The resulting wire bundle encodes all and only the occurrences of labels within the expression's syntax tree, which have to be correct with respect to T. Now that we have a formal definition, we can define what it means for a Proto-Quipper-RA judgment, and by extension for an RMI, to be correct.

Definition 6.2 (Correct Judgment). We say that a judgment $\emptyset; \Xi. \bullet; Q \vdash^c_{\mathfrak{t},\mathfrak{c}} M : A; I$ is *correct* when for all C such that $\Xi \vdash^s_{\mathfrak{c}} C : L \to Q, H; J$ there exist $\mathcal{D}, V, K, E$ such that

(1) $(C, M) \Downarrow (C :: \mathcal{D}, V)$,
(2) $\Xi \vdash^s_{\mathfrak{c}} \mathcal{D} : Q \to K; E$,
(3) $\vDash_{\mathfrak{t},\mathfrak{c}} E \leq I$,
(4) $LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(K, V, A)$.

Definition 6.3 (Correct RMI). We say that an RMI $\mathfrak{t}$ is *correct* with respect to a CMI $\mathfrak{c}$ when every judgment of the form $\emptyset; \Xi. \bullet; Q \vdash^c_{\mathfrak{t},\mathfrak{c}} M : A; I$ derivable in Proto-Quipper-RA is correct.

For simplicity, we say that the Proto-Quipper-RA instance defined by RMI $\mathfrak{t}$ and CMI $\mathfrak{c}$ is correct when $\mathfrak{t}$ is correct with respect to $\mathfrak{c}$.

6.2 Semantic Interpretation of Types

In order to prove correctness, we rely on the logical relations technique [63]. The first thing we have to do is provide a *semantic interpretation* of types and effects. This interpretation characterizes what it means for a value or term to *semantically* be of a certain type and/or have a certain effect. For the sake of clarity, when reasoning in these terms, we explicitly keep track of which free labels occur in a value or term, by means of a label context Q. The formal definition of the semantic interpretation of types and effects is thus as follows.

Definition 6.4 (Semantic Interpretation of Types). We define $\mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A] \subseteq \mathscr{Q} \times \mathit{VAL}$ and $\mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A;I] \subseteq \mathscr{Q} \times \mathit{TERM}$ as the smallest type-indexed relations such that the following conditions hold:

$$
\begin{aligned}
(\bullet, *) &\in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\mathbb{1}],\\
(\bullet, \mathsf{nil}) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\mathsf{List}_{i<I}\, A] &\Leftrightarrow \ \vDash I = 0,\\
(\ell : w^I, \ell) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[w^J] &\Leftrightarrow \Xi \vDash_{\mathfrak{t},\mathfrak{c}} I \le J,\\
(\bullet, \mathsf{lift}\, M) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[!^I A] &\Leftrightarrow (\bullet, M) \in \mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A;I],\\
(Q \uplus L, \langle V, W\rangle) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A \otimes B] &\Leftrightarrow ((Q,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]) \wedge ((L,W) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[B]),\\
(Q \uplus L, \mathsf{rcons}\ V\ W) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\mathsf{List}_{i<I}\, A] &\Leftrightarrow \exists J.((\vDash I = J+1) \wedge ((Q,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\mathsf{List}_{i<J}\, A])\\
&\qquad \wedge ((L,W) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A\{J/i\}])),\\
(Q, \lambda x_A.M) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A \multimap^I_T B] &\Leftrightarrow ((Q, \mathit{labs}(M)) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[T]) \wedge \forall L, V.(((L,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A])\\
&\qquad \Rightarrow ((Q \uplus L, M[V/x]) \in \mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[B;I])),\\
(\bullet, (\bar{\ell}, C, \bar{k})) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\mathsf{Circ}^I_\Delta(T,U)] &\Leftrightarrow (\Delta \vdash^s_{\mathfrak{c}} C : Q \to L; J)\\
&\qquad \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} J \le I) \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} \#(Q) \le I) \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} \#(L) \le I)\\
&\qquad \wedge ((Q, \bar{\ell}) \in \mathfrak{V}^{\Delta}_{\mathfrak{t},\mathfrak{c}}[T]) \wedge ((L, \bar{k}) \in \mathfrak{V}^{\Delta}_{\mathfrak{t},\mathfrak{c}}[U]),\\
(Q, M) \in \mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A;I] &\Leftrightarrow \forall C.((\Xi \vdash^s_{\mathfrak{c}} C : L \to Q, H; J) \Rightarrow \exists \mathcal{D}, V, K, E.\\
&\qquad (((C,M) \Downarrow (C :: \mathcal{D}, V)) \wedge (\Xi \vdash^s_{\mathfrak{c}} \mathcal{D} : Q \to K; E)\\
&\qquad \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} E \le I) \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} \#(Q) \le I) \wedge (\vDash_{\mathfrak{t},\mathfrak{c}} \#(K) \le I)\\
&\qquad \wedge ((K,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]))).
\end{aligned}
$$

Note that because Q and A may naturally contain free variables encoding local metrics, the interpretation relation is annotated with the index context Ξ from which these variables come from. This specific approach to logical relations, where a semantic interpretation is naturally given to non-ground types, is known as open logical relations [5]. Predicate $\mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]$ mostly follows the structure of types. Predicate $\mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A;I]$, on the other hand, tells us what we expect from the execution of a term M of type A and effect annotation I. Intuitively, it tells us that whenever we pair M with a circuit C that supplies the wires in Q, M evaluates to V and extends C with a subcircuit $\mathcal{D}$ with specific properties. Namely, $\mathcal{D}$'s size, as well as the size of its inputs and outputs, are upper-bounded by I. Furthermore, the labels output by $\mathcal{D}$, i.e. those in K, are all and only the free labels occurring in V, and together with these labels V is semantically of type A.

Note that these definitions only apply to expressions that are closed in the sense of regular variables. In order to deal with open expressions, we extend the semantic interpretation of types to typing contexts. Informally, a typing context Γ is interpreted by a value substitution γ which substitutes each x in Γ with a value V that is semantically of type $\Gamma(x)$. More formally:

Definition 6.5 (Semantic Interpretation of Typing Contexts). We define $\mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\Gamma] \subseteq \mathscr{Q} \times \mathit{VSUB}$ as the smallest context-indexed relation such that the following conditions hold:

$$(\bullet, \emptyset) \in \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\bullet], \qquad (Q \uplus L, \gamma[x \mapsto V]) \in \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\Gamma, x : A] \Leftrightarrow (Q,\gamma) \in \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\Gamma] \wedge (L,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A],$$

where, for all $e \in \mathit{VAL} \cup \mathit{TERM}$, $\emptyset(e) = e$ and $\gamma[x \mapsto V](e) = \gamma(e[V/x])$.

Now we have all the ingredients to define the semantic well-typedness of (possibly open) values and terms. Recall that $\vartheta \in \mathfrak{I}_{\emptyset}[\Theta]$ means that index substitution ϑ substitutes each index variable in Θ with a closed index. We have the following definition:

Definition 6.6 (Semantic Well-typedness). If for all $\vartheta \in \mathfrak{I}_\emptyset[\Theta]$ and all $(L,\gamma) \in \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\vartheta(\Gamma)]$ we have

(1) $(Q \uplus L, \gamma(\vartheta(M))) \in \mathfrak{T}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\vartheta(A);\vartheta(I)]$, then we write $\Theta;\Xi : \Gamma; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$,
(2) $(Q \uplus L, \gamma(\vartheta(V))) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\vartheta(A)]$, then we write $\Theta;\Xi : \Gamma; Q \vDash^{v}_{\mathfrak{t},\mathfrak{c}} V : A$.

6.3 Proving Correctness

Next, we show that Proto-Quipper-RA typing entails semantic typing, and that semantic typing in turn entails correctness. Naturally, this proof would be impossible to carry out without imposing some constraints on the RMI defining the derivable judgments. We therefore identify a small set of constraints on RMIs, which we call *coherence*, that are sufficient to ensure that an RMI is correct. These constraints relate an RMI to an underlying CMI, and ensure that the former is indeed a sound overapproximation of the latter.

Definition 6.7 (Coherence). We say that an RMI $\mathfrak{t}$ is *coherent* with a CMI $\mathfrak{c}$ when

(1) $\mathfrak{size}^{\emptyset}_{\mathfrak{c}} \leq \mathfrak{e}_{\mathfrak{t}}$ and $\mathfrak{size}^{\widetilde{Q}+\{w\}}_{\mathfrak{c}} \leq \mathfrak{par}_{\mathfrak{t}}(\mathfrak{size}^{\widetilde{Q}}_{\mathfrak{c}}, \mathfrak{wire}^{w}_{\mathfrak{t}})$, for all $Q \in \mathscr{Q}, w \in \{\mathsf{Qubit}, \mathsf{Bit}\}$,
(2) $\mathfrak{append}^{g}_{\mathfrak{c}}(\mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, n), \mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, m), i, o) \leq \mathfrak{par}_{\mathfrak{t}}(\mathfrak{wire}^{w}_{\mathfrak{t}}, \mathfrak{append}^{g}_{\mathfrak{c}}(n, m, i, o))$, for all $g \in \mathscr{G}, w \in \mathscr{W}, n, m, i, o \in \mathbb{N}$,
(3) $\mathfrak{append}^{g}_{\mathfrak{c}}(\mathfrak{seq}_{\mathfrak{t}}(p, n), m, i, o) \leq \mathfrak{seq}_{\mathfrak{t}}(p, \mathfrak{append}^{g}_{\mathfrak{c}}(n, m, i, o))$, for all $g \in \mathscr{G}, p, n, m, i, o \in \mathbb{N}$.

The first constraint ensures that the type size function induced by $\mathfrak{t}$ correctly overapproximates the size of identity circuits, i.e. that for all $Q \in \mathscr{Q}$ we have $\vDash_{\mathfrak{t},\mathfrak{c}} \#(Q) \leq \widetilde{Q}$. Constraint (2) is a sort of lax distributive law, telling us that factoring $\mathfrak{par}_{\mathfrak{t}}$ over $\mathfrak{append}^{g}_{\mathfrak{c}}$ yields a sound overapproximation. Conversely, constraint (3) tells us that if a gate is appended to the composition in sequence of two circuits C and $\mathcal{D}$, we can overapproximate the overall size by considering the composition in sequence of C and g appended to $\mathcal{D}$. For simplicity, we say that a Proto-Quipper-RA instance is coherent when its RMI is coherent with its CMI. Under this condition, we can show that well-typedness in Proto-Quipper-RA implies semantic well-typedness.

Lemma 6.8. *Let RMI $\mathfrak{t}$ be coherent with CMI $\mathfrak{c}$. Let Π be a type derivation. We have that*

$$\Theta;\Xi.\,\Gamma; Q \vdash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I \Rightarrow \Theta;\Xi : \Gamma; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I,$$
$$\Theta;\Xi.\,\Gamma; Q \vdash^{v}_{\mathfrak{t},\mathfrak{c}} V : A \Rightarrow \Theta;\Xi : \Gamma; Q \vDash^{v}_{\mathfrak{t},\mathfrak{c}} V : A.$$

Proof. By induction on the size of Π. □

Next, it is easy to show that semantic well-typedness implies correctness both in the sense of global resources and of local resources, by which we conclude the main correctness claim.

Lemma 6.9. *If $\emptyset;\Xi : \bullet; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$, then for every C such that $\Xi \vDash^{s}_{\mathfrak{t},\mathfrak{c}} C : L \to Q, H; J$ there exist $\mathcal{D}, V, K, E$ such that*

$$(C, M) \Downarrow (C :: \mathcal{D}, V) \wedge \Xi \vDash^{s}_{\mathfrak{t},\mathfrak{c}} \mathcal{D} : Q \to K; E \wedge \ \vDash_{\mathfrak{t},\mathfrak{c}} E \leq I \wedge LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(K, V, A).$$

Proof. By the definition of $\emptyset;\Xi : \bullet; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$ we get the first three clauses and $(K,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]$. It remains to show that $(K,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]$ implies $LRC^{\Xi}_{\mathfrak{t},\mathfrak{c}}(K,V,A)$, which is easy by induction on the proof of $(K,V) \in \mathfrak{V}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[A]$. □

Theorem 6.10 (Total Correctness). *Every coherent Proto-Quipper-RA instance is correct.*

Proof. Let $\mathfrak{t}$ be an RMI that is coherent with a CMI $\mathfrak{c}$ and let $\emptyset;\Xi. \bullet; Q \vdash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$ be a derivable judgment in Proto-Quipper-RA. By Lemma 6.8, since $\emptyset \in \mathfrak{I}_\emptyset[\emptyset]$ and $(\bullet, \emptyset) \in \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\bullet] = \mathfrak{C}^{\Xi}_{\mathfrak{t},\mathfrak{c}}[\emptyset(\bullet)]$, we get $\emptyset;\Xi : \bullet; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$. In turn, by Lemma 6.9, $\emptyset;\Xi : \bullet; Q \vDash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$ entails that

$\emptyset; \Xi. \bullet; Q \vdash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$ is correct. Since we imposed no constraints on $\emptyset; \Xi. \bullet; Q \vdash^{c}_{\mathfrak{t},\mathfrak{c}} M : A; I$, we get that every judgment of that form derivable in the given Proto-Quipper-RA instance is correct, and thus the instance is itself correct. □

7 Implementing Proto-Quipper-RA within QuRA

QuRA is a tool for the analysis of the size of the circuits produced by quantum circuit description programs.[2] QuRA is implemented in Haskell and is originally based on the theoretical framework presented in [8]. As such, it is originally limited to the estimation of only the width of the quantum circuits produced by the programs. More specifically, QuRA takes as input a program written in a concrete dialect of the Proto-Quipper language and returns the type of the program, alongside the width of the circuit it builds. QuRA relies on a type inference algorithm and on SMT-solving [6] to infer this information automatically, with minimal program annotations.

Our contribution involves a substantial generalization and extension of QuRA's resource estimation framework, based on the theoretical work presented in this paper. First of all, we used Proto-Quipper-RA's type-and-effect system as a basis for the generalization of QuRA's inference algorithm: at each inference step, instead of synthesizing an index describing *specifically* the width of the constructed circuit, the algorithm now synthesizes an index using abstract resource composition operations. Whenever indices need to be checked, these operators are translated into different arithmetic operations depending on the definition of the global metric under analysis. This definition is independent from the main inference logic, being implemented in an external *global metric module* that closely resembles an RMI. Next, we extended QuRA's input language with the ability to specify *local* metrics, and we updated the inference algorithm accordingly. Like in the global case, inference is agnostic to the concrete local metric being analyzed, which is provided externally through a *local metric module*. Having done this, we were able to reimplement width estimation as a special case of the abstract framework, through the definition of an appropriate module. In addition to width, we were also able to easily implement a variety of new metric modules, thus extending QuRA with the ability of analyzing metrics such as gate count, depth, T-count, T-depth and more. Note that these modules are relatively small, averaging at around 40 lines of Haskell code each, and they completely encapsulate the logic concerning the estimation of the corresponding metric. This makes it particularly easy for external contributors to extend QuRA with support for new metrics, or to tune existing metric definitions to fit specific needs.

7.1 Some Examples of Automatic Resource Estimation

Using our extension of QuRA, we were able to analyze a number of quantum algorithms for their resource consumption, including the Quantum Fourier Transform (QFT) algorithm [11] and Grover's algorithm [32]. In this section, we use those as case studies to illustrate what kind of analyses can be carried out through our generic approach to resource estimation.

7.1.1 The Quantum Fourier Transform. The program in Figure 13 implements the QFT algorithm, a fundamental subroutine in several quantum algorithms. The programming language used is the input language to QuRA, and it is not too dissimilar from the language described in Section 5, with a few notable differences: the syntax is plaintext and Haskell-like; index variables are explicitly scoped and instantiated, with `forall` acting as a variable binder in both terms and types and `@` denoting index application; and type refinements are enclosed within either square or curly brackets, depending on whether they concern global or local metrics. For example, `List[j<n] Qubit` corresponds to $\mathsf{List}_{j<n}\ \mathsf{Qubit}$, while `Qubit{d}` corresponds to Qubit^{d}. We assume that `qrev` is a function that reverses a list of qubits, while `cR @i @d1 @d2` applies a controlled rotation (whose

[2] https://github.com/andreacolledan/qura

```
1  -- apply the cR gate with target trg ad depth d during iteration iter
2  let rotate = forall d. forall iter. lift forall step.
3    \((ctrls, trg), ctrl) :: ((List[i<step] Qubit{d+iter+i+1},
4                                Qubit{d+iter+step}), Qubit{d+iter+step}).
5      let (ctrl, trg) = (force cr @(iter+1-step) @(d+iter+step) @(d+iter+step))
           ctrl trg in (ctrls:ctrl, trg)
6  in
7  -- apply the Quantum Fourier Transform to n qubits at depth d
8  let qft = forall n. forall d. \reg :: List[i<n] Qubit{d}.
9    let qftIter = lift forall iter. -- define the iteration of the QFT
10     \(ctrls, trg)::(List[i<iter] Qubit{d+iter+i}, Qubit{d}).
11       let revctrls = (force qrev @iter @d) ctrls in
12       let (ctrls, trg) = fold(rotate @d @iter, ([], trg), revctrls) in
13       let trg = (force hadamard @(d+2*iter)) trg in ctrls:trg
14   in fold(qftIter, [], reg)
15 in qft
```

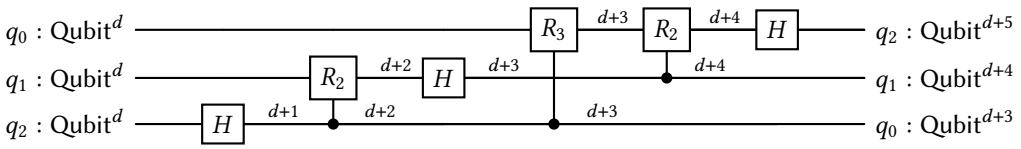

Fig. 13. An implementation of the QFT algorithm and the corresponding circuit for $n = 3$. Labels show the depth of the wire segments.

magnitude depends on i) to a pair of qubits at depths d_1 and d_2, respectively. The full code of the QFT is available in QuRA's repository, under `examples`.

When the program is fed to QuRA, we can decide to just type-check it, or to also statically estimate the metrics associated with the circuits it produces. At the time of writing, QuRA supports the simultaneous analysis of at most one global and one local metric, so for example we can analyze the width and depth of the circuits produced by `qft` as follows:

```
$ qura examples/qft.pqr -g width -l depth
* Inferred type: forall[0,0] n. forall[0,0] d. List[i<n] Qubit{d}
    -o[n,0] List[i<n] Qubit{d+n+i}
```

Depth-wise, this type tells us that the `qft` function takes as input a list of n qubits at depth d and outputs n qubits such that the i-th qubit sits at depth $d + n + i$. Width-wise, the first annotation on the arrow tells us that the function builds a circuit of width at most n. These bounds are not only sound, but also exact, as can be seen in the case of the QFT circuit of input size 3 shown in Figure 13. If we were interested in the gate count of the circuits produced by `qft`, we could run

```
$ qura examples/qft.pqr -g gatecount -l depth
* Inferred type: forall[0,0] n. forall[0,0] d. List[i<n] Qubit{d}
    -o[sum[iter<n](iter+1),0] List[i<n] Qubit{d+n+i}
```

This tells us that on an input of size n, the `qft` function builds a circuit comprising of at most $\sum_{iter=0}^{n-1} iter + 1$ gates, which is once again an exact estimate.

7.1.2 Grover's Algorithm. The program in Figure 14 implements Grover's search algorithm. We omit the definition of the `diffusion` function for brevity, but we remark that `diffusion @n @d` applies Grover's diffusion operator to n qubits and one ancilla qubit, all at depth d. From a resource analysis perspective, the interesting thing about this algorithm is that it does not take as input a

```
1  -- perform a single grover iteration on n qubits at depth d, using an ancilla
       a at depth d and an oracle of size os and overall depth od
2  let groverIteration = lift forall n. forall d. forall os. forall od.
3    \oracle :: forall[0,0] d. Circ[os](
4                                (List[_<n] Qubit{d}, Qubit{d}),
5                                (List[_<n] Qubit{d+od}, Qubit{d+od})).
6    \reg :: List[_<n] Qubit{d}. \a :: Qubit{d}.
7      let (reg, a) = apply(oracle @ d, (reg, a)) in
8      (force diffusion @ n @ d + od) reg a
9  in
10 -- run Grover's algorithm for r iterations. Oracle parameters are n, os and od
11 let grover = forall r. forall n. forall os. forall od.
12   \oracle :: forall[0,0] d. Circ[os](
13                               (List[_<n] Qubit{d}, Qubit{d}),
14                               (List[_<n] Qubit{d+od}, Qubit{d+od})).
15     -- prepare n working qubits and one ancilla
16     let wqs = force qinitMany0 @ n in
17     let wqs = (force mapHadamard @ n @ 0) wqs in
18     let a = force qinit1 in
19     let a = (force hadamard @ 0) a in
20     -- instantiate parameters for groverIteration and iterate
21     let iteration = lift forall step. \((wqs, a), _) :: ((List[_<n] Qubit{1+
           step*(od+5)}, Qubit{1+step*(od+5)}), ()).
22       (force groverIteration @ n @ 1+step*(od+5) @ os @ od) oracle wqs a
23     in
24     let (wqs, a) = fold(iteration, (wqs, a), range @ r) in
25     let _ = (force qdiscard @ 1+r*(od + 5)) a in
26     ((force mapMeasure @ n @ 1+r*(od + 5)) wqs :: List[_<n] Bit{2+r*(od+5)})
27 in grover
```

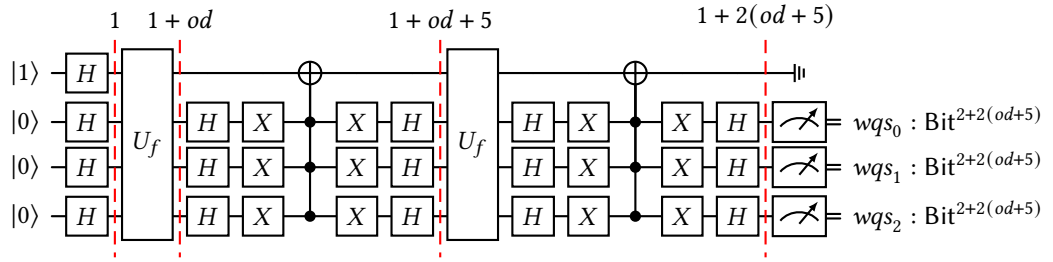

Fig. 14. An implementation of Grover's algorithm and the corresponding circuit for a 3-qubit oracle U_f of depth od. Slices show the overall depth at key points in the circuit.

register, but rather an *oracle*, that is, a circuit implementing a function to be queried. In our case, this entails that the `grover` function that implements the algorithm is higher-order, as it takes as input an oracle in the form of a boxed circuit whose type is shown in lines 12 to 14. The parameters of the algorithm are thus the number of iterations r and the parameters of the oracle: its input size n, its circuit size os and its overall depth od.

The width analysis of `grover` is fairly uninteresting, as it is simply the size of the oracle's input plus one ancilla, so we focus on a gate count and depth analysis instead. Each Grover iteration consists in one oracle invocation, followed by the application of the diffusion operator. By definition, the oracle has gate count os and depth od, while the diffusion operator has gate count $4n + 1$ and depth 5. Therefore, each iteration has a gate count of $os + 4n + 1$ and a depth of $od + 5$. With the

$n + 1$ Hadamard gates used to intialize the state, r Grover iterations, and n final measurements, we reach a total gate count of $2n + 1 + r(os + 4n + 1)$ and an overall depth of $2 + r(od + 5)$. Can QuRA derive the same concrete upper bounds? The answer is affirmative:

```
$ qura examples/grover.pqr -g gatecount -l depth
* Inferred type: forall[0,0] r. forall[0,0] n. forall[0,0] os. forall[0,0] od.
  (forall[0,0] d. Circ[os](
    (List[_<n] Qubit{d}, Qubit{d}), (List[_<n] Qubit{d+od}, Qubit{d+od}))
  ) -o[n+1 + r*(os + 4*n + 1) + n, 0] List[_<n] Bit{2 + r*(od+5)}
```

The first annotation on the linear arrow tells us that the gate count of the circuits produced by `grover` is bounded by $n + 1 + r(os + 4n + 1) + n = 2n + 1 + r(os + 4n + 1)$, while the wire type annotation on the output tells us that all of the output bits are at depth of at most $2 + r(od + 5)$. As an aside, note that since r is usually chosen to be close to $\frac{\pi}{4}\sqrt{2^n}$, our depth estimate is consistent with the asymptotic time complexity of Grover's algorithm, which is $O(\sqrt{2^n})$.

8 Related Work

Proto-Quipper-RA is by no means the first paradigmatic language derived from `Quipper`. Some members of the Proto-Quipper family model the basic features of the language [57, 58], while others model its extension with dynamic lifting [7, 24, 46] or dependent types [25]. The family member that has influenced this work the most is certainly Proto-Quipper-R [8]. However, as we have already pointed out, Proto-Quipper-R's analysis is limited to circuit width, a metric which, although crucial, is not always the most informative in terms of resource consumption.

Resource analysis for quantum programs has been advocated as crucial by the security community, where quantum attacks are considered a threat and it is therefore crucial to understand, for instance, how many qubits are needed to implement certain attacks (see, e.g., [2, 29]). This type of analysis, however, is usually performed on an individual circuit and not parametrically on the size of the input, which is what our work instead achieves. The complexity analysis of quantum programs, on the other hand, has been the subject of recent investigation, with tools coming for example from the so-called implicit computational complexity [15, 68], but also through tools such as weakest precondition calculi [4, 50]. Overall, however, the literature is much more sparse in the quantum case than it is in the classical, deterministic, but even probabilistic case, where existing contributions range from type systems [21] to amortized analysis [38], from weakest precondition calculi [43] to abstract interpretation [1]. Among these approaches, the ones closest to ours are those of Dal Lago, Gaboardi, and co-authors [3, 13, 16, 26], which are sometimes referred to as *linear dependent types*. Besides the treatment of the peculiarities of Proto-Quipper, the most important novelty of our contribution compared to the aforementioned works lies in the fact that index terms are not used to control duplication (in the spirit of graded comonads) but rather to control the size of the produced circuit. The latter can, depending on the nature of the underlying metric, become a global parameter, in the spirit of graded monads, or remain local to each individual wire. Another line of work that employs tools similar to ours is that of resource analysis based on session types [14, 18–20]. Here, arithmetic expressions are included in the underlying linear type theory to represent a notion of size, in the spirit of sized types [39, 45]. In the present work, however, we focus on a specific data structure and a peculiar notion of computation, namely quantum circuits. It must be noted that although these circuits can effectively be seen as classical data structures, and although `Quipper`'s operational semantics is classical, the way `Quipper` generates circuits is still peculiar, due to the fact that the construction of circuits is an effect, that wires must be treated linearly, and that circuits can be boxed and unboxed (i.e. that effects can become data, and vice versa).

As an aside, note that optimizing the width of quantum circuits via qubit recycling is a hard problem [42]. Certainly, we cannot say that our type system gives an estimate of the underlying circuit's *optimized form*, something which would be quite difficult to do via types anyway. At the same time, if the programmer takes the responsibility of correctly using recycling as a means to natively describe circuits of optimal width, then our resource analysis is able to reflect the improvement in the inferred bounds. This is because we employ a *concrete* circuit model, which can at least *express* recycling, unlike the traditional categorical model given by Selinger and Rios [57].

On the side of *functional* properties, the verification of of quantum programs has also received some attention by the research community [47]. Various kinds of verification problems have been considered, including termination [48, 49], robustness [33, 40], equivalence checking [66], and the correctness of compiler optimizations [37] using tools like Hoare logics [35], interactive theorem proving [36], and type systems [64]. Note, however, that most these works are concerned with *imperative* quantum programming languages.

Finally, on the side of classical programming, it is worth mentioning that refinement types have been successfully used to perform resource analysis in `Liquid Haskell` [34]. However, the lack of closure types [59] and coeffects [54] in `Haskell`, together with inadequate support for linear types, makes it difficult to apply the techniques from [34] to the problem analyzed in this paper.

9 Conclusion and Future Work

In this work we presented for the first time a type system capable of deriving upper bounds on the size of the quantum circuits produced by programs in the `Quipper` language. Crucially, the upper bounds we get are not constant, but rather parametric on classical circuit-building parameters, such as the size of the input. Furthermore, the metric used to measure the size of circuits is not fixed, but can in fact be any metric whose semantic interpretation satisfies some reasonable conditions.

We also used our theoretical framework to massively generalize and extend the capabilities of the `QuRA` tool, demonstrating how several parametric resource analyses can be defined easily and carried out on real-world quantum algorithms in a fundamentally automatic fashion.

As for future work, it would be interesting to take into consideration metrics that are not natural-valued. Although not necessarily a *resource* metric, we believe that a functional property like *circuit accuracy* could be captured in our framework as a local metric. This would of course require extending the language of indices so as to include real numbers, and adapting the surrounding theory accordingly. On the side of `QuRA`, benchmarks of the introduced technique, which so far are limited to a modest set of programs, are also something we plan to work on in the near future. Finally, on the metatheoretical side, there is certainly an open question regarding whether sufficient conditions exist that can ensure the derivability of a suitable interpretation for a certain metric. In other words, it would be interesting to know that certain circuit metrics can be analyzed in our type system without having to explicitly construct a corresponding CMI or RMI. A denotational account of the semantics of Proto-Quipper-RA is also a topic of future work.

Acknowledgments

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".

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2008. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. of FMCO 2007*. 113–132. https://doi.org/10.1007/978-3-540-92188-2_5

[2] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. 2017. Estimating the Cost of Generic Quantum Pre-image Attacks on SHA-2 and SHA-3. In *Proc. of SAC 2016*. 317–337. https://doi.org/10.1007/978-3-319-69453-5_18

[3] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Proc. of LICS 2019*. 1–13. https://doi.org/10.1109/LICS.2019.8785725

[4] Martin Avanzini, Georg Moser, Romain Pechoux, Simon Perdrix, and Vladimir Zamdzhiev. 2022. Quantum Expectation Transformers for Cost Analysis. In *Proc. of LICS 2022*. 1–13. https://doi.org/10.1145/3531130.3533332

[5] Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. 2020. On the Versatility of Open Logical Relations. In *Proc. of ESOP 2020*. 56–83. https://doi.org/10.1007/978-3-030-44914-8_3

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2021. *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. https://doi.org/10.3233/FAIA336

[7] Andrea Colledan and Ugo Dal Lago. 2023. On Dynamic Lifting and Effect Typing in Circuit Description Languages. In *Proc. of TYPES 2022*, Vol. 269. 3:1–3:21. https://doi.org/10.4230/LIPIcs.TYPES.2022.3

[8] Andrea Colledan and Ugo Dal Lago. 2024. Circuit Width Estimation via Effect Typing and Linear Dependency. In *Proc. of ESOP 2024*. 3–30. https://doi.org/10.1007/978-3-031-57267-8_1

[9] Hugh Collins and Chris Nay. 2022. IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum System Two. https://is.gd/WPV7lO Retrieved on Nov. 10, 2024.

[10] Emily Conover. 2020. Light-based Quantum Computer Jiuzhang achieves quantum supremacy. https://is.gd/zIgFzK retrieved on Nov. 10, 2024.

[11] D. Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring, IBM Research Report RC19642. arXiv:quant-ph/0201067

[12] 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, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *TQC* 3, 3 (2022), 1–50. https://doi.org/10.1145/3505636

[13] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proc. of LICS 2011*. 133–142. https://doi.org/10.1109/LICS.2011.22

[14] Ugo Dal Lago and Giulia Giusti. 2022. On Session Typing, Probabilistic Polynomial Time, and Cryptographic Experiments. In *Proc. of CONCUR 2022*, Vol. 243. 37:1–37:18. https://doi.org/10.4230/LIPIcs.CONCUR.2022.37

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

[16] Ugo Dal Lago and Barbara Petit. 2012. Linear Dependent Types in a Call-by-Value Scenario. In *Proc. of PPDP 2012*. 115–126. https://doi.org/10.1145/2370776.2370792

[17] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *Proc. of POPL 2013*. 167–178. https://doi.org/10.1145/2429069.2429090

[18] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.* 2, ICFP (2018), 1–30. https://doi.org/10.1145/3236786

[19] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *Proc. of LICS 2018*. 305–314. https://doi.org/10.1145/3209108.3209146

[20] Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. In *Proc. of CONCUR 2020*, Vol. 171. 13:1–13:18. https://doi.org/10.4230/LIPIcs.CONCUR.2020.13

[21] Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. In *Proc. of POPL 2023*. 32 pages. https://doi.org/10.1145/3571259

[22] Cirq Developers. 2024. *Cirq*. https://doi.org/10.5281/zenodo.11398048

[23] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proc. of PLDI 1991*. 268–277. https://doi.org/10.1145/113445.113468

[24] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2023. Proto-Quipper with Dynamic Lifting. In *Proc. of POPL 2023*. 309–334. https://doi.org/10.1145/3571204

[25] Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract. In *Proc. of LICS 2020*. 440–453. https://doi.org/10.1145/3373718.3394765

[26] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proc. of POPL 2013*. 357–370. https://doi.org/10.1145/2429069.2429113

[27] Simon J. Gay. 2006. Quantum Programming Languages: Survey and Bibliography. *Math. Struct. Comput. Sci.* 16, 4 (2006), 581–600. https://doi.org/10.1017/S0960129506005378

[28] Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlote, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, Kamalpreet Kaur, Priyansh Arora, Ji Liu, Ahmed Farouk, Houbing Herbert Song, Steve Uhlig, and Kotagiri Ramamohanarao. 2024. Quantum Computing: Vision and Challenges. arXiv:2403.02240

[29] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. 2016. Applying Grover's Algorithm to AES: Quantum Resource Estimates. In *Proc. of PQCrypto 2016*. 29–43. https://doi.org/10.1007/978-3-319-29360-8_3
[30] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper. In *Proc. of PLDI*. 333–342. https://doi.org/10.1145/2499370.2462177
[31] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proc. of PLDI 2013*. 333–342. https://doi.org/10.1145/2491956.2462177
[32] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. arXiv:arXiv:quant-ph/9605043
[33] Ji Guan, Wang Fang, and Mingsheng Ying. 2021. Robustness Verification of Quantum Classifiers. In *Proc. of CAV 2021*, Vol. 12759. 151–174. https://doi.org/10.1007/978-3-030-81685-8_7
[34] Martin Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate your assets: reasoning about resource usage in liquid Haskell. *PACMPL* 4 (2019), 1–27. https://doi.org/10.1145/3371092
[35] Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-based optimization of Quantum programs. In *Proc. of OOPSLA 2020*. 1–20. https://doi.org/10.1145/3428201
[36] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. In *Proc. of ITP 2021*, Vol. 193. 21:1–21:19. https://doi.org/10.4230/LIPICS.ITP.2021.21
[37] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. In *Proc. of POPL 2021*. 1–29. https://doi.org/10.1145/3434318
[38] Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487
[39] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Proc. of POPL 1996*. 410–423. https://doi.org/10.1145/237721.240882
[40] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative robustness analysis of quantum programs. In *Proc. of POPL 2019*. 31:1–31:29. https://doi.org/10.1145/3290344
[41] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. arXiv:2405.08810
[42] Hanru Jiang. 2024. Qubit Recycling Revisited. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1264–1287. https://doi.org/10.1145/3656428
[43] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Proc. of ESOP 2016*. 364–389. https://doi.org/10.1007/978-3-662-49498-1_15
[44] E. Knill. 2022. Conventions for Quantum Pseudocode. arXiv:2211.02559
[45] Ugo Dal Lago and Charles Grellois. 2019. Probabilistic Termination by Monadic Affine Sized Typing. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 1–65. https://doi.org/10.1145/3293605
[46] 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 2021*, Vol. 213. 51:1–51:20. https://doi.org/10.4230/LIPIcs.FSTTCS.2021.51
[47] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. 2023. Formal Verification of Quantum Programs: Theory, Tools, and Challenges. *TQC* 5, 1 (2023), 1–35. https://doi.org/10.1145/3624483
[48] Yangjia Li and Mingsheng Ying. 2018. Algorithmic analysis of termination problems for quantum programs. In *Proc. of POPL 2018*. 35:1–35:29. https://doi.org/10.1145/3158123
[49] Yangjia Li, Nengkun Yu, and Mingsheng Ying. 2014. Termination of nondeterministic quantum programs. *Acta Informatica* 51, 1 (2014), 1–24. https://doi.org/10.1007/S00236-013-0185-3
[50] 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*. 1–13. https://doi.org/10.1145/3531130.3533327
[51] John Martinis. 2019. Quantum supremacy using a programmable superconducting processor. https://is.gd/v3VXFi Retrieved on Nov. 10, 2024.
[52] Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2016. Effect Systems Revisited—Control-Flow Algebra and Semantics. In *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. 1–32. https://doi.org/10.1007/978-3-319-27810-0
[53] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design: Recent Insights and Advances*. Springer Berlin Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7
[54] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *Proc. of ICFP 2014*. 123–135. https://doi.org/10.1145/2628136.2628160
[55] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[56] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *JACM* 56, 6 (2009), 1–40. https://doi.org/10.1145/1568318.1568324
[57] Francisco Rios and Peter Selinger. 2017. A categorical model for a quantum circuit description language. In *Proc. of QPL 2017*, Vol. 266. 164–178. https://doi.org/10.4204/EPTCS.266.11
[58] Neil Ross. 2015. *Algebraic and Logical Methods in Quantum Computation.* Ph. D. Dissertation. Dalhousie University.
[59] Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *Proc. of LPAR 2013*. 710–726. https://doi.org/10.1007/978-3-642-45221-5_47
[60] Maximilian Schlosshauer. 2007. *Decoherence and the Quantum-To-Classical Transition.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-35775-9
[61] Peter Selinger. 2004. A Brief Survey of Quantum Programming Languages. In *Proc. of FLOPS 2004*. 1–6. https://doi.org/10.1007/978-3-540-24754-8_1
[62] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proc. of FOCS 1994*. 124–134. https://doi.org/10.1109/sfcs.1994.365700
[63] Lau Skorstengaard. 2019. An Introduction to Logical Relations. arXiv:1907.11133
[64] Aarthi Sundaram, Robert Rand, Kartik Singhal, and Brad Lackey. 2022. A Rich Type System for Quantum Programs. arXiv:2101.08939
[65] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proc. of ICFP 2014*. 269–282. https://doi.org/10.1145/2628136.2628161
[66] Qisheng Wang, Junyi Liu, and Mingsheng Ying. 2021. Equivalence checking of quantum finite-state machines. *J. Comput. Syst. Sci.* 116 (2021), 1–21. https://doi.org/10.1016/J.JCSS.2020.08.004
[67] Peter Wittek. 2014. *Quantum machine learning: what quantum computing means to data mining.* Academic Press.
[68] Tomoyuki Yamakami. 2020. A Schematic Definition of Quantum Polynomial Time Computability. *J. Symb. Log.* 85, 4 (2020), 1546–1587. https://doi.org/10.1017/jsl.2020.45