Sized Types for low-level Quantum Metaprogramming

Matthew Amy[1][0000−0003−3514−420X]

University of Waterloo, Waterloo, Canada
meamy@uwaterloo.ca

Abstract. One of the most fundamental aspects of quantum circuit design is the concept of *families* of circuits parametrized by an instance size. As in classical programming, metaprogramming allows the programmer to write entire families of circuits simultaneously, an ability which is of particular importance in the context of quantum computing as algorithms frequently use arithmetic over non-standard word lengths. In this work, we introduce metaQASM, a typed extension of the openQASM language supporting the metaprogramming of circuit families. Our language and type system, built around a lightweight implementation of *sized types*, supports subtyping over register sizes and is moreover type-safe. In particular, we prove that our system is strongly normalizing, and as such any well-typed metaQASM program can be statically unrolled into a finite circuit.

Keywords: Quantum programming, Circuit description languages, Metaprogramming.

1 Introduction

Quantum computers have the potential to solve a number of important problems, including integer factorization [29], quantum simulation [23], approximating the Jones polynomial [1] and unstructured searching [12] asymptotically faster than the best known classical algorithms. These algorithms are typically described abstractly and make heavy use of classical arithmetic such as modular exponentiation. To make such algorithms concrete, efficient, reversible implementations of large swaths of a classical arithmetic and computation is needed – moreover, due to the limited space constraints and special-purpose nature of quantum circuits, these operations are typically needed in a multitude of bit sizes.

In part due to the increasing viability of quantum computing and the scaling of NISQ [28] devices, there has been a recent explosion in quantum programming tools. Such tools range from software development kits (e.g., Qiskit [5], ProjectQ [32], Strawberry Fields [19], Pyquil [30]) to Embedded domain-specific languages (e.g., Quipper [11], Qwire [27], $Q|SI\rangle$ [22]) and standalone languages and compilers (e.g., QCL [26], QML [2], ScaffCC [16], Q# [33]). Going beyond strict programming tools, software for the synthesis, optimization, and simulation of quantum circuits and

programs (e.g., Revkit [31], TOpt [15], Feynman [3], PyZX [20], Quantum++ [9], QX [17]) are becoming more and more abundant.

The proliferation of both hardware and software tools for quantum computing has in turn spurred a need for standardization and portability [14,24]. One such standard which has recently grown in popularity is the Quantum Assembly Language and its many various dialects (e.g., openQASM [6], QASM-HL [16], cQASM [18]). As a lightweight, modular language for specifying simple quantum circuits, programs with a well-defined syntax, QASM support – in particular, for the openQASM dialect – has been built-in to an increasingly large number of software tools, particularly standalone programs like circuit optimizers, as a way to support interoperability.

One feature that is noticeably lacking in these dialects is the ability to define *families* of quantum circuits parametrized over different register sizes, and by extension to *generate* concrete instances. This creates a barrier for the use of QASM in writing portable libraries of quantum circuit families, particularly for classical operations such as arithmetic. As a result, software designers typically end up re-implementing code – typically implemented in the host language for EDSLs, and hence not easily re-usable – for generating instances of simple operations such as adders and multipliers. Alternatively, programmers resort to using other compilers such as Quipper, Q# or ReVerC [4] to generate individual instances, which complicates the compilation or simulation process. While recent progress towards the development of portable libraries of circuit families with high-level non-embedded languages, standardization remains an ongoing process, and moreover a low-level approach is preferable in many situations, including as compilation targets and middle-ends.

In this paper we make progress towards the design of a low-level language for quantum programming that supports the metaprogramming of sized circuit families. In particular, we develop a typed extension of the untyped open quantum assembly language (openQASM) with metaprogramming over lightweight *sized types* à la dependent ML [34]. Our language, metaQASM, is further shown to be type-safe and strongly-normalizing, while the non-meta fragment is both more expressive than openQASM and admits a simpler syntax, owing to the type system. For the purposes of this paper, we focus on the type system design and metatheory of such a language, leaving implementation to future work.

1.1 Quantum metaprogramming

Most QRAM-based quantum programming languages are metaprogramming languages – called *circuit description languages* – in that they typically operate by building quantum circuits to be sent in a single batch to a quantum processor. Such quantum circuits can typically be composed, reversed, and depend on the result of classical computations.

In this paper, we are interested in a particular type of quantum circuit metaprogramming, wherein circuit families are parametrized over *shapes* [11,27], such as the number of input qubits. Existing languages offer varying support for such metaprogramming, either implicitly (e.g., uniform or *transversal* families of circuits in openQASM, iteration and qubit arrays

in Q#), or more explicitly (e.g., the generic `QData` type-class in Quipper, which can be instantiated via explicit type applications). Our approach differs from previous attempts by explicitly parametrizing registers and circuit families with *size* parameters. We adopt a typed approach for a number of reasons:

- it allows the light-weight verification of libraries of circuit generators,
- it provides a means of self-documentation, and
- it allows explicit generation of sized-specialized instances.

The ability to generate instances of circuit families in various sizes *without executing them* is particularly important for the purposes of resource estimation, and for benchmarking tools that operate on fixed-size but arbitrary input circuits, such as circuit optimizers [14].

As an illustration, given an in-place family of adders written in the style of (imperative) Quipper with the type

```
inplace_add :: [Qubit] -> [Qubit] -> Circ (),
```

one may wish to generate a static, optimized instance of `inplace_add` operating on 2-qubit registers, using an external circuit optimizer. Doing so requires the specialization to (and serialization of) a function

```
inplace_add2 :: (Qubit, Qubit) -> (Qubit, Qubit) -> Circ ().
```

One possible method of generating such a function is to write the body of `inplace_add2` using a call to the generic `inplace_add` applied to the 4 input qubits. However, this quickly gets unwieldy, both in the boilerplate code defining a particular instance, and in the large number of parameters.

A more common solution is to use *dummy parameters*, whereby the generic function is "applied" to lists of qubits, which are then taken by the serialization method as meaning arbitrary inputs. For instance, the following Quipper[1] code [10] prints out a PDF representation of `inplace_add2` using dummy parameters `qubit :: Qubit`

```
print_generic PDF inplace_add [qubit, qubit] [qubit, qubit].
```

The use of dummy parameters is partly a question of style, though it can cause problems when combining optimizations with *initialized* dummy parameters. In either case, the use explicitly sized circuit families carries further benefits to both readability and correctness [27].

1.2 Organization

The remainder of this paper is organized as follows. Section 2 gives a brief overview of quantum computing. Section 3 reviews the openQASM language and defines a formal semantics for it. Sections 4 and 5 extend openQASM with types and metaprogramming capabilities, and finally Section 6 concludes the paper.

---

[1] The function `inplace_add2` could instead be directly generated by writing the adder as `inplace_add :: QData qa => qa -> qa -> Circ ()`, then specializing `qa` to the finite type `(Qubit, Qubit)` using *type applications*. However, the non-generic serialization functions in Quipper appear to work only for small finite tuple types.

2   Quantum computing

We give a brief overview of the basics of quantum computing. For a more in-depth introduction of quantum computation we direct the reader to [25], while an overview of quantum programming can be found in [8].

In the circuit model, the state of an n-qubit quantum system is described as a unit vector in a dimension 2^n complex vector space. The 2^n elementary basis vectors form the *computational* basis, and are denoted by $|\mathbf{x}\rangle$ for bit strings $\mathbf{x} \in \{0,1\}^n$ – these are called the *classical* states. A general quantum state may then be written as a *superposition* of classical states

$$|\psi\rangle = \sum_{\mathbf{x} \in \mathbb{F}_2^n} \alpha_{\mathbf{x}} |\mathbf{x}\rangle,$$

for complex $\alpha_{\mathbf{x}}$ and having unit norm. The states of two n and m qubit quantum systems $|\psi\rangle$ and $|\psi\rangle$ may be combined into an $n+m$ qubit state by taking their tensor product $|\psi\rangle \otimes |\psi\rangle$. If to the contrary the state of two qubits cannot be written as a tensor product the two qubits are said to be *entangled*.

Quantum circuits, in analogy to classical circuits, carry qubits from left to right along *wires* through *gates* which transform the state. In the unitary circuit model gates are required to implement unitary operators on the state space – that is, quantum gates are modelled by complex-valued matrices U satisfying $UU^\dagger = U^\dagger U = I$, where $U^\dagger$ is the complex conjugate of U. As a result, unitary quantum computations must be *reversible*, and in particular the quantum circuits performing classical computations are precisely the set of reversible circuits.

The standard universal quantum gate set, known as Clifford+T, consists of the two-qubit controlled-NOT gate (CNOT), and the single-qubit Hadamard (H) and T gates. As quantum circuits implement linear operators, we may define the above three gates by their effect on classical states:

$$\text{CNOT}|x\rangle|y\rangle = |x\rangle|x \oplus y\rangle, \qquad T|x\rangle = e^{\frac{2\pi i}{8}x}|x\rangle,$$

$$H|x\rangle = \frac{1}{\sqrt{2}} \sum_{x' \in \{0,1\}} (-1)^{x \cdot x'} |x'\rangle.$$

Figure 1 gives a pictorial representation of a quantum circuit over CNOT, H, and T gates. CNOT gates are written as a solid dot on their first argument and an exclusive-OR symbol ($\oplus$) on their second argument.

More general quantum operations include qubit initialization and measurement, which effectively convert between classical and quantum data. As neither operation is unitary and hence not (directly) reversible, we regard them as functions of the classical computer rather than gates in a quantum circuit.
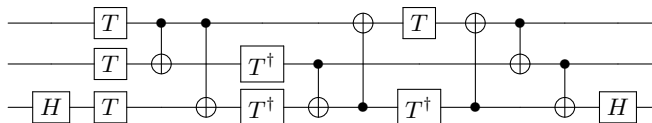

Fig. 1. An example of a quantum circuit implementing the Toffoli gate.

3   openQASM

The open quantum assembly language (openQASM [6]) is a low-level, untyped imperative quantum programming language, developed as a dialect of the informal QASM language. One of the key additions of the openQASM language is that of *modularity*, in the form of a simple module and import system. As this work is largely concerned with the question of *making this modularity more powerful* – specifically, to support the modular definition of entire circuit families – we first give a brief overview of the openQASM language.

The official specification of openQASM can be found in [6]. Programs in openQASM are structured as sequences of declarations and commands. Programmers can declare statically-sized classical or quantum registers, define unitary circuits (called *gates* in openQASM), apply gates or circuits, measure or initialize qubits and condition commands on the value of classical bits. Gate arguments are restricted to individual qubits, where the application of gates to one or more register *of the same size* is syntactic sugar for the application of a single gate in parallel across the registers. The listing below gives an example of an openQASM program performing quantum teleportation:

```
OPENQASM 2.0;
qreg q[3];
creg c0[1];
creg c1[1];

h q[1];
cx q[1],q[2];
cx q[0],q[1];
h q[0];
measure q[0] -> c0[0];
measure q[1] -> c1[0];
if(c0==1) z q[2];
if(c1==1) x q[2];
```

We give a slightly different syntax from the above, and from the concrete syntax [6], as it will be more convenient and readable for our purposes. As is common in imperative languages, we leave some of the concrete syntactic classes of openQASM [6] separate in our formalization – since all operations in openQASM nominally have unit type, this allows terms with unitary and non-unitary *effects* to be distinguished, without relying on an effect system or monadic types. In particular, terms of the class U of unitary statements represent computations with purely unitary effects, while commands C may have non-unitary effects, such as measurement. Statements of the form

$$E(E_1, \ldots, E_n)$$

represent the application of a unitary gate or named circuit E to the (quantum) arguments E_1 through E_n. While the openQASM specification includes built-in cx (controlled-NOT) and parametrized single qubit

$$
\begin{aligned}
\text{Identifier } & x \\
\text{Index } I & ::= i \in \mathbb{N} \\
\text{Expression } E & ::= x \mid x[I] \\
\text{Unitary Stmt } U & ::= \mathtt{cx}(E_1, E_2) \mid \mathtt{h}(E) \mid \mathtt{t}(E) \mid \mathtt{tdg}(E) \mid E(E_1, \ldots, E_n) \mid U_1;\ U_2 \\
\text{Command } C & ::= \mathtt{creg}\ x[I] \mid \mathtt{qreg}\ x[I] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \\
& \quad \mid \mathtt{measure}\ E_1\ \mathtt{->}\ E_2 \mid \mathtt{reset}\ E \mid U \\
& \quad \mid \mathtt{if}(E\mathtt{==}I)\ \{\ U\ \} \mid C_1;\ C_2
\end{aligned}
$$

$$
\begin{aligned}
\text{Location } l & \in \mathbb{N} \\
\text{Value } V & ::= (l_0, \ldots, l_{I-1}) \mid \lambda x_1, \ldots, x_n.U
\end{aligned}
$$

Fig. 2. openQASM (abstract) syntax

gates U, we drop the parametrized U gate in favour of built-in Hadamard and $T/T^{\dagger}$ gates $\mathtt{h}$ and $\mathtt{t}/\mathtt{tdg}$, respectively.

The commands $\mathtt{creg}$, $\mathtt{qreg}$ and $\mathtt{gate}$ declare classical registers, quantum registers, and unitary circuits, respectively. The $\mathtt{if}$ statement differs from the formal openQASM definition by testing the value of a *single* classical bit, rather than a classical register – this was done to simplify the semantics of the language. Locations l and values V do not appear directly in openQASM programs, but are used to define the semantics. In particular, values of the form $(l_0, \ldots, l_{I-1})$ denote registers and $\lambda x_1, \ldots, x_n.U$ denote unitary circuits. We leave out a number of features of openQASM which are orthogonal to the extensions we describe here, namely classical arithmetic and the $\mathtt{barrier}$ and $\mathtt{opaque}$ terms. We also write parentheses around arguments and parameters.

As no formal semantics of openQASM is given in [6], we define an operational semantics in Figure 3. Our semantics is defined with respect to a *configuration* $\langle S, \sigma, \eta, |\psi\rangle \rangle$, which stores a term S taken from some syntactic class (e.g., C, U, E), an environment σ which maps variables to values, a classical heap η storing the value of the classical bits, and a quantum state $|\psi\rangle$. Gates applied to qubit l of a quantum state are written by added a subscript to the intended gate, e.g.,

$$
H_l|\psi\rangle = (I^{\otimes l-1} \otimes H \otimes I^{\otimes n-l})|\psi\rangle
$$

$\sigma[x \leftarrow v]$ denotes the environment mapping x to v or $\sigma(x)$ otherwise, and $S\{X/x\}$ denotes the substitution of X for x in S. We assume for convenience that no valid program will run out of classical memory or quantum bits. We say $\langle S, \sigma, \eta, |\psi\rangle \rangle \Downarrow v$ if S reduces to v, where the form of v depends on the syntactic class of S – for instance, expressions evaluate to locations, arrays or circuits while commands produce a new environment, heap and quantum state. Note that we use a call-by-name evaluation strategy, as openQASM has only globally scoped variables.

Rather than give a full probabilistic reduction system to account for measurement probabilities, it suffices for our purposes to make the semantics non-deterministic. In particular, rules are given for both of the possible measurement outcomes in $\mathtt{measure}\ E_1\ \mathtt{->}\ E_2$, setting the classical bit to the result $c \in \{0, 1\}$ and non-destructively applying the projector $P^c = |c\rangle\langle c|$ (appropriately normalized) to the measured qubit.

Expressions:

$$\frac{x \in \mathsf{dom}(\sigma)}{\langle x, \sigma, \eta, |\psi\rangle\rangle \Downarrow \sigma(x)} \qquad \frac{\langle x, \sigma, \eta, |\psi\rangle\rangle \Downarrow (l_0, \ldots, l_{I'}) \qquad I \leq I'}{\langle x[I], \sigma, \eta, |\psi\rangle\rangle \Downarrow l_I}$$

Unitary statements:

$$\frac{\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l}{\langle \mathtt{h}(E), \sigma, \eta, |\psi\rangle\rangle \Downarrow H_l|\psi\rangle} \qquad \frac{\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l}{\langle \mathtt{t}(E), \sigma, \eta, |\psi\rangle\rangle \Downarrow T_l|\psi\rangle} \qquad \frac{\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l}{\langle \mathtt{tdg}(E), \sigma, \eta, |\psi\rangle\rangle \Downarrow T_l^\dagger|\psi\rangle}$$

$$\frac{\langle E_1, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_1 \quad \langle E_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_2}{\langle \mathtt{cx}(E_1, E_2), \sigma, \eta, |\psi\rangle\rangle \Downarrow \mathrm{CNOT}_{l_1, l_2}|\psi\rangle} \qquad \frac{\begin{array}{c}\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow \lambda x_1, \ldots, x_n.U, \\ \langle U\{E_1/x_1, \ldots, E_n/x_n\}, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle\end{array}}{\langle E(E_1, \ldots, E_n), \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle}$$

$$\frac{\langle U_1, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle \qquad \langle U_2, \sigma, \eta, |\psi'\rangle\rangle \Downarrow |\psi''\rangle}{\langle U_1;\ U_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi''\rangle}$$

Commands:

$$\frac{l_0, \ldots, l_{I-1} \text{ are fresh heap indices}}{\langle \mathtt{creg}\ x[I], \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma[x \leftarrow (l_0, \ldots, l_{I-1})], \eta, |\psi\rangle\rangle}$$

$$\frac{l_0, \ldots, l_{I-1} \text{ are fresh qubit indices}}{\langle \mathtt{qreg}\ x[I], \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma[x \leftarrow (l_0, \ldots, l_{I-1})], \eta, |\psi\rangle\rangle}$$

$$\overline{\langle \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \}, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma[x \leftarrow \lambda x_1, \ldots, x_n.U], \eta, |\psi\rangle\rangle}$$

$$\frac{\langle E_1, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_1 \qquad \langle E_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_2}{\langle \mathtt{measure}\ E_1\ \texttt{->}\ E_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma, \eta[l_2 \leftarrow 0], P_{l_1}^0|\psi\rangle\rangle}$$

$$\frac{\langle E_1, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_1 \qquad \langle E_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_2}{\langle \mathtt{measure}\ E_1\ \texttt{->}\ E_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma, \eta[l_2 \leftarrow 1], P_{l_1}^1|\psi\rangle\rangle}$$

$$\frac{\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l}{\langle \mathtt{reset}\ E, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma, \eta, P_l^0|\psi\rangle\rangle} \qquad \frac{\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l \qquad \eta(l) \neq I}{\langle \mathtt{if(E==I)}\ \{\ U\ \}, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma, \eta, |\psi\rangle\rangle}$$

$$\frac{\begin{array}{c}\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow l \qquad \eta(l) = I \\ \langle U, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle\end{array}}{\langle \mathtt{if(E==I)}\ \{\ U\ \}, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma, \eta, |\psi'\rangle\rangle}$$

$$\frac{\langle C_1, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma', \eta', |\psi'\rangle\rangle \qquad \langle C_2, \sigma', \eta', |\psi'\rangle\rangle \Downarrow \langle \sigma'', \eta'', |\psi''\rangle\rangle}{\langle C_1;\ C_2, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma'', \eta'', |\psi''\rangle\rangle}$$

Fig. 3. openQASM semantics

Base types $\beta ::= \texttt{Bit} \mid \texttt{Qbit}$
     Types $\tau ::= \beta \mid \beta[I] \mid \texttt{Circuit}(\tau_1, \ldots, \tau_n)$
Command $C ::= \ldots \mid \texttt{creg } x[I] \texttt{ in } \{\ C\ \} \mid \texttt{qreg } x[I] \texttt{ in } \{\ C\ \}$
               $\mid \texttt{gate } x(x_1 : \tau_1, \ldots, x_n : \tau_n) \{\ U\ \} \texttt{ in } \{\ C\ \}$

Fig. 4. typedQASM specification

4   Adding types to QASM

Run-time errors may occur in syntactically valid openQASM programs
in a number of ways – particularly when either an array access is out of
bounds and the program halts, or a classical (resp. quantum) location is
used in a context when a quantum (resp. classical) location is expected.
In the official openQASM specification, the latter error is eliminated by
the requirement that only (global) variables can be declared as quantum
registers may be used as arguments to gates, for instance. In either case
however, it is desirable to check that an openQASM program *will not
go wrong*, as circuit simulations are frequently run on large, expensive
supercomputers (e.g., [13]).
In this section we developed a typed variant of openQASM, called type-
dQASM, which provably rules out such runtime errors. Moreover, the
type system uses *sized types* to eliminate out-of-bound accesses, which
we later develop into the core of our metaprogramming type system. The
use of a type system in this case actually allows *more* valid programs to
be written than the standard openQASM specification, as the type sys-
tem allows us to remove some syntactic distinctions and instead make
them in the type system. In particular, our type system allows regis-
ters and circuits to be passed as functions to other circuits, whereas the
formal specification restricts circuit arguments to only individual qubits.
Figure 4 gives the syntax of typedQASM. We only show the syntactic
elements which are different from openQASM or otherwise new. To sim-
plify our analysis, declarations are given explicit block scope, though we
leave textual examples in the regular openQASM style of declaration. As
the semantics of typedQASM is effectively identical, modulo the block
scoping, to openQASM we don't explicitly give the semantics.

4.1   The type system

Figure 5 gives the rules of our type system. As is standard, the judgement
$\Gamma \vdash S : \tau$ states that in the context Γ consisting of pairs of identifiers
and types, S can be assigned type τ. We overload $\vdash$ to allow environment
judgements of the form $\vdash \sigma : \Gamma$ stating that the σ maps identifiers x to
values of the type τ if $x : \tau \in \Gamma$.
The type system of typedQASM is mostly as expected, with the excep-
tion of static-length registers and register bounds checks in the typing
rules for dereferences. To give the programmer flexibility to apply gates
and circuits to just parts of a larger register – for instance, when perform-
ing an n-bit addition into a length $2n$ register as in binary multiplication

Environment:

$$\frac{}{\vdash \cdot : \cdot} \qquad \frac{\vdash \sigma : \Gamma}{\vdash \sigma[x \leftarrow (l_0, \ldots, l_{I-1})] : \Gamma, x : \beta[I]}$$

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash U : \mathtt{Unit}}{\vdash \sigma[x \leftarrow \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U] : \Gamma, x : \mathtt{Circuit}(\tau_1, \ldots, \tau_n)}$$

Expressions:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash x : \beta[I'] \qquad I \leq I' - 1}{\Gamma \vdash x[I] : \beta} \qquad \frac{\Gamma \vdash E : \beta[I'] \qquad I \leq I'}{\Gamma \vdash E : \beta[I]}$$

Unitary statements:

$$\frac{\Gamma \vdash E_1 : \mathtt{Qbit} \qquad \Gamma \vdash E_2 : \mathtt{Qbit}}{\Gamma \vdash \mathtt{cx}(E_1, E_2) : \mathtt{Unit}} \qquad \frac{\Gamma \vdash E : \mathtt{Qbit} \qquad g \in \{\mathtt{h}, \mathtt{t}, \mathtt{tdg}\}}{\Gamma \vdash g(E) : \mathtt{Unit}}$$

$$\frac{\Gamma \vdash E : \mathtt{Circuit}(\tau_1, \ldots, \tau_n)}{\Gamma \vdash E_1 : \tau_1 \quad \cdots \quad \Gamma \vdash E_n : \tau_n}{\Gamma \vdash E(E_1, \ldots, E_n) : \mathtt{Unit}} \qquad \frac{\Gamma \vdash U_1 : \mathtt{Unit} \qquad \Gamma \vdash U_2 : \mathtt{Unit}}{\Gamma \vdash U_1\texttt{;}\ U_2 : \mathtt{Unit}}$$

Commands:

$$\frac{\Gamma, x : \mathtt{Bit}[I] \vdash C : \mathtt{Unit}}{\Gamma \vdash \mathtt{creg}\ x[I]\ \mathtt{in}\ \{\ C\ \} : \mathtt{Unit}} \qquad \frac{\Gamma, x : \mathtt{Qbit}[I] \vdash C : \mathtt{Unit}}{\Gamma \vdash \mathtt{qreg}\ x[I]\ \mathtt{in}\ \{\ C\ \} : \mathtt{Unit}}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash U : \mathtt{Unit} \quad \Gamma, x : \mathtt{Circuit}(\tau_1, \ldots, \tau_n) \vdash C : \mathtt{Unit}}{\Gamma \vdash \mathtt{gate}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \{\ U\ \}\ \mathtt{in}\ \{\ C\ \} : \mathtt{Unit}}$$

$$\frac{\Gamma \vdash E_1 : \mathtt{Qbit} \qquad \Gamma \vdash E_2 : \mathtt{Bit}}{\Gamma \vdash \mathtt{measure}\ E_1\ \texttt{->}\ E_2 : \mathtt{Unit}} \qquad \frac{\Gamma \vdash E : \mathtt{Qbit}}{\Gamma \vdash \mathtt{reset}\ E : \mathtt{Unit}}$$

$$\frac{\Gamma \vdash E : \mathtt{Bit} \qquad \Gamma \vdash U : \mathtt{Unit}}{\Gamma \vdash \mathtt{if(}E\texttt{==}I\mathtt{)}\ \{\ U\ \} : \mathtt{Unit}} \qquad \frac{\Gamma \vdash C_1 : \mathtt{Unit} \qquad \Gamma \vdash C_2 : \mathtt{Unit}}{\Gamma \vdash C_1\texttt{;}\ C_2 : \mathtt{Unit}}$$

Fig. 5. typedQASM typing rules

– the type system also implicitly supports subtyping of static length registers. Specifically, any length I array can be used in a context requiring *at most I* cells. While this adds a great deal of flexibility on the side of the programmer, as a downside typedQASM typing derivations are not unique.

As an example of a well-typed QASM program, we show an implementation of the Toffoli circuit from Figure 1 below:

```
gate toffoli(x:Qbit, y:Qbit, z:Qbit) {
    h(z);
    t(x); t(y); t(z);
    cx(x,y); cx(x,z);
    tdg(y); tdg(z);
    cx(y,z); cx(z,x);
    t(x); tdg(z);
    cx(z,x); cx(x,y); cx(y,z);
    h(z)
}
```

4.2 Type safety

We now briefly sketch a proof of type safety for typedQASM. In particular, we show that typedQASM is strongly normalizing, as expected.

As is standard, we establish strong normalization by giving type preservation and progress lemmas. While type preservation is effectively implicit in the semantics of typedQASM due to the different syntactic classes, expressions may return different types of values and so we give a form of type preservation for such terms.

Lemma 1 (Preservation (expressions)). *If $\Gamma \vdash E : \tau$, $\vdash \sigma : \Gamma$ and $\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow v$, then either*

- *$\tau = \beta$ and $v = l$ for some base type β & location l,*
- *$\tau = \beta[I]$ and $v = (l_0, \ldots, l_{I'})$ where $I' \geq I$, or*
- *$\tau = \mathtt{Circuit}(\tau_1, \ldots, \tau_n)$ and $v = \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U$.*

Proof. If $\tau = \beta$ then we must have $E = x[I]$, hence by the definition of $\Downarrow$, $v = l$. Likewise if $\tau = \beta[I]$ then we must have $E = x$ where $x : \beta[I'] \in \Gamma$ for some $I' \geq I$, and since $\vdash \sigma : \Gamma$ then $v = \sigma(x) = (l_0, \ldots, l_{I'})$. The case for $\tau = \mathtt{Circuit}(\tau_1, \ldots, \tau_n)$ is similar.

The following lemmas give progress properties – the fact that for a well-typed program, evaluation can always continue – for the different syntactic classes of typedQASM. Together with type preservation, the result is that any well-typed typedQASM program evaluates to a value, i.e. that typedQASM is strongly normalizing.

Lemma 2 (Progress (expressions)). *If $\Gamma \vdash E : \tau$ and $\vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow v$.*

Proof. By case analysis on E. If $E = x$ the proof is trivial, as $x : \tau \in \Gamma$ by inversion and $\vdash \sigma : \Gamma$ implies $x \in \mathsf{dom}(x)$. If on the other hand $E = x[I]$, we must have $x : \beta[I'] \in \Gamma$ for some $I' > I$. Then by preservation, $\langle x, \sigma, \eta, |\psi\rangle\rangle \Downarrow (l_0, \ldots, l_{I''})$ for some $I'' \geq I' - 1$, hence $\langle x, \sigma, \eta, |\psi\rangle\rangle \Downarrow l_I$

Lemma 3 (Progress (unitary stmts)). *If $\Gamma \vdash U : \mathtt{Unit}$ and $\vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle U, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle$.*

Proof. For the case $U = E(E_1, \ldots, E_n)$, by the typing derivation we have $\Gamma \vdash E : \mathtt{Circuit}(\tau_1, \ldots, \tau_n)$ so by progress and preservation for expressions, $\langle E, \sigma, \eta, |\psi\rangle\rangle \Downarrow \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U$. By the substitution lemma below, $\Gamma \vdash U\{E_1/x_1, \ldots, E_n/x_n\} : \mathtt{Unit}$ and hence we can structural induction to show that $\langle U, \sigma, \eta, |\psi\rangle\rangle \Downarrow |\psi'\rangle$.

Lemma 4 (Substitution). *If $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash U : \mathtt{Unit}$, and $\Gamma \vdash E_i : \tau_i$ for each $1 \leq i \leq n$ then $\Gamma \vdash U\{E_1/x_1, \ldots, E_n/x_n\} : \mathtt{Unit}$*

Lemma 5 (Progress (commands)). *If $\Gamma \vdash C : \mathtt{Unit}$ and $\vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle C, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma', \eta', |\psi'\rangle\rangle$.*

Proof. Proof by induction on the structure of C. We show one case:

$$C = \texttt{gate } x(x_1 : \tau_1, \ldots, x_n : \tau_n) \; \{ \; U \; \} \; \texttt{in} \; \{ \; C \; \}$$

We know that

$$\langle \texttt{gate } x(x_1 : \tau_1, \ldots, x_n : \tau_n) \; \{ \; U \; \}, \sigma, \eta, |\psi\rangle \rangle \Downarrow \langle \sigma[x \leftarrow \lambda x_1, \ldots, x_n.U], \eta, |\psi\rangle \rangle.$$

By the typing derivation, $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash U : \texttt{Unit}$ and $\Gamma, x : \texttt{Circuit}(\tau_1, \ldots, \tau_n) \vdash C : \texttt{Unit}$. It then follows that

$$\vdash \sigma[x \leftarrow \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U] : \Gamma, x : \texttt{Circuit}(\tau_1, \ldots, \tau_n),$$

and hence we can apply the inductive hypothesis to complete the case. The remaining cases are similar.

Theorem 1 (Strong normalization). *If* $\vdash C : \texttt{Unit}$, *then*

$$\langle C, \emptyset, \lambda l.0, |00 \cdots\rangle \rangle \Downarrow \langle \sigma, \eta, |\psi\rangle \rangle.$$

Proof. Direct consequence of Lemma 5.

5   MetaQASM

Now that we have a safe, array-bounds-checked, typed language, we can add metaprogramming features. In particular, we wish to support[2]
– circuit inversion/reversal, and
– circuits parametrized by sizes.
While the latter could be accomplished in an ad-hoc way, allowing *type-level* integers allows for more safety in that array bounds can be statically checked, and increases the readability of programs. Moreover, it enforces a clear separation between circuits and families of circuits, which naturally support different operations – for instance, a family of circuits can't easily be visualized diagrammatically, while a particular instance can [27].
Figure 6 gives the new syntax for metaQASM. Indices I are extended with index variables y and integer arithmetic, and a new syntactic form defining a family of quantum circuits parametrized over index variables is given. The index ∞ only exists in the process of type checking and is not valid syntax in source code. Intuitively, the declaration

$$\texttt{family}(y_1, \ldots, y_m) \; x(x_1 : \tau_1, \ldots, x_n : \tau_n) \; \{ \; U \; \} \; \texttt{in} \; \{ \; C \; \}$$

introduces index variables $y_1, \ldots, y_m$ into the evaluation and type checking contexts for τ_i and U.

---

[2] Controlled circuits are another desirable metaprogramming feature found in many quantum circuit description languages. While metaQASM gates are in fact closed over qubit controls, they require *ancillae* to construct [21]. This complicates the inclusion of a control instruction in metaQASM, and further abstracts away from concrete, resource-driven nature of QASM.

$$
\begin{aligned}
\text{Types } \tau &::= \ldots \mid \texttt{Family}(y_1, \ldots, y_m)(\tau_1, \ldots, \tau_n) \\
\text{Index } I &::= \ldots \mid y \mid \infty \mid I_1 + I_2 \mid I_1 - I_2 \mid I_1 \cdot I_2 \\
\text{Range } \iota &::= [I_1, I_2] \\
\text{Expression } E &::= \ldots \mid \texttt{instance}(I_1, \ldots, I_m)\ E \\
\text{Unitary Stmt } U &::= \ldots \mid \texttt{reverse}\ U \mid \texttt{for}\ y = I_1..I_2\ \texttt{do}\ \{\ U\ \} \\
\text{Command } C &::= \ldots \mid \texttt{family}(y_1, \ldots, y_m)\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \{\ U\ \}\ \texttt{in}\ \{\ C\ \} \\
\text{Value } V &::= \ldots \mid \Pi y_1, \ldots, y_m . V
\end{aligned}
$$

Fig. 6. metaQASM syntax

Figure 7 gives the semantics of the new syntax. Since index variables cannot be modified or captured, we use a substitution style of evaluation for circuit families. The $\texttt{reverse}$ command introduces a new reduction relation $\langle U, \sigma, |\psi\rangle \rangle \Uparrow v$ for which reduction of U is inverted. We give a concrete semantics rather than an abstract rule such as

$$
\frac{\langle U, \sigma, \eta, |\psi'\rangle \rangle \Downarrow |\psi\rangle}{\langle \texttt{reverse}\ U, \sigma, \eta, |\psi\rangle \rangle \Downarrow |\psi'\rangle}
$$

so that metaQASM has a concrete execution model. Inversion of circuits is straightforward in metaQASM, as in any closed context a unitary statement can be statically unrolled to a finite sequence of gates.

As an illustration of metaprogramming in metaQASM, Figure 8 gives metaQASM code for a simple (non-garbage-cleaning) adder. Our syntax (and type system) also allows an instance of a family of circuits to accept other circuit families as arguments, a useful feature which allows circuit families to be parametric in the implementation of a sub-routine as shown below (using a minor syntax extension to allow array slicing).

```
family(n) mult(x:Qbit[n], y:Qbit[n], z:Qbit[2*n],
    anc:Qbit, ctrlAdd:Family(m)
        (x:Qbit, y:Qbit[m], z:Qbit[m], c:Qbit))
{
  for i=0..n-1 do {
    instance(n) ctrlAdd(x[i], y, z[i..i+n-1], anc)
  }
}
```

By extending our syntax with parametrized gates as in regular open-QASM, we can also define a parametrized family of circuits computing the *quantum Fourier transform* as in [27].

```
include "cphase.qasm";
family(n) qft(x:Qbit[n]) {
    for i=0..n-1 do {
        h(x[i]);
        for j=i+1..n-1 do {
            cphase(j-1+1)(x[i], x[j])
        }
    }
}
```

Indices:

$$\frac{}{\langle i,\sigma,\eta,|\psi\rangle\rangle \Downarrow i} \qquad \frac{\langle I_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \quad \langle I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_2 \quad \star \in \{+,-,\cdot\}}{\langle I_1 \star I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \star i_2}$$

Expressions:

$$\frac{\langle E,\sigma,\eta,|\psi\rangle \Downarrow \Pi y_1,\ldots,y_m.\lambda x_1:\tau_1,\ldots,x_n:\tau_n.U}{\langle \mathtt{instance}(I_1,\ldots,I_m)\ E,\sigma,\eta,|\psi\rangle \Downarrow (\lambda x_1:\tau_1,\ldots,x_n:\tau_n.U)\{I_1/y_1,\ldots,I_m/y_m\}}$$

Unitary statements:

$$\frac{\langle U,\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle}{\langle \mathtt{reverse}\ U,\sigma,\eta,|\psi\rangle\rangle \Downarrow |\psi'\rangle} \qquad \frac{\langle I_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \quad \langle I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_2 \quad i_1 > i_2}{\langle \mathtt{for}\ y = I_1..I_2\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi\rangle\rangle \Downarrow |\psi\rangle}$$

$$\frac{\langle I_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \quad \langle I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_2 \quad i_1 \leq i_2}{\langle U\{i_1/y\},\sigma,\eta,|\psi\rangle\rangle \Downarrow |\psi'\rangle \qquad \langle \mathtt{for}\ y = i_1+1..i_2\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi'\rangle\rangle \Downarrow |\psi''\rangle}{\langle \mathtt{for}\ y = I_1..I_2\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi\rangle\rangle \Downarrow |\psi''\rangle}$$

Reverse reduction:

$$\frac{\langle E,\sigma,\eta,|\psi\rangle\rangle \Downarrow l}{\langle \mathtt{h}(E),\sigma,\eta,|\psi\rangle\rangle \Uparrow H_l|\psi\rangle} \qquad \frac{\langle E,\sigma,\eta,|\psi\rangle\rangle \Downarrow l}{\langle \mathtt{t}(E),\sigma,\eta,|\psi\rangle\rangle \Uparrow T_l^\dagger|\psi\rangle} \qquad \frac{\langle E,\sigma,\eta,|\psi\rangle\rangle \Downarrow l}{\langle \mathtt{tdg}(E),\sigma,\eta,|\psi\rangle\rangle \Uparrow T_l|\psi\rangle}$$

$$\frac{\langle E_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow l_1 \quad \langle E_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow l_2}{\langle \mathtt{cx}(E_1,E_2),\sigma,\eta,|\psi\rangle\rangle \Uparrow \mathrm{CNOT}_{l_1,l_2}|\psi\rangle} \qquad \frac{\langle E,\sigma,\eta,|\psi\rangle\rangle \Downarrow \lambda x_1,\ldots,x_n.U,\ \langle U\{E_1/x_1,\ldots,E_n/x_n\},\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle}{\langle E(E_1,\ldots,E_n),\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle}$$

$$\frac{\langle U_2,\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle \qquad \langle U_1,\sigma,\eta,|\psi'\rangle\rangle \Uparrow |\psi''\rangle}{\langle U_1;\ U_2,\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi''\rangle}$$

$$\frac{\langle U,\sigma,\eta,|\psi\rangle\rangle \Downarrow |\psi'\rangle}{\langle \mathtt{reverse}\ U,\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle} \qquad \frac{\langle I_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \quad \langle I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_2 \quad i_2 < i_1}{\langle \mathtt{for}\ y = I_1..I_2\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi\rangle}$$

$$\frac{\langle I_1,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_1 \quad \langle I_2,\sigma,\eta,|\psi\rangle\rangle \Downarrow i_2 \quad i_2 \geq i_1}{\langle U\{i_2/y\},\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi'\rangle \qquad \langle \mathtt{for}\ y = i_1..i_2-1\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi'\rangle\rangle \Uparrow |\psi''\rangle}{\langle \mathtt{for}\ y = I_1..I_2\ \mathtt{do}\ \{\ U\ \},\sigma,\eta,|\psi\rangle\rangle \Uparrow |\psi''\rangle}$$

Commands:

$$\frac{\langle C,\sigma[x \leftarrow \Pi y_1,\ldots,y_m.\lambda x_1:\tau_1,\ldots,x_n:\tau_n.U],\eta,|\psi\rangle\rangle \Downarrow \langle \sigma',\eta',|\psi'\rangle\rangle}{\langle \mathtt{family}(y_1,\ldots,y_m)\ x(x_1:\tau_1,\ldots,x_n:\tau_n)\ \{\ U\ \}\ \mathtt{in}\ \{\ C\ \},\sigma,\eta,|\psi\rangle\rangle \Downarrow \langle \sigma,\eta',|\psi'\rangle\rangle}$$

Fig. 7. metaQASM semantics

```
include "toffoli.qasm";
gate maj(a:Qbit, b:Qbit, c:Qbit, res:Qbit) {
    toffoli(b, c, res);
    cx(b, c);
    toffoli(a, c, res);
    cx(b, c)
}
family(n) add(a:Qbit[n], b:Qbit[n], c:Qbit[n], anc:Qbit[n]) {
    cx(a[0], c[0]);
    cx(b[0], c[0]);
    toffoli(a[0], b[0], anc[0]);
    for i=1..n-1 do {
        cx(a[i], c[i]);
        cx(b[i], c[i]);
        cx(anc[i-1], c[i]);
        maj(a[i], b[i], anc[i-1], anc[i])
    }
}
```

Fig. 8. metaQASM implementation of a carry-ripple adder.

5.1 Type system

The type system of metaQASM is inspired by Dependent ML [34]. Figure 9 gives the rules of our system. Type rules are defined over two contexts $\Delta; \Gamma$, where Δ contains interval constraints on index variables. As with typedQASM, array bounds are checked and subtyping on array lengths is allowed. Integer expressions are assigned intervals which may be arbitrary (well-formed) integer expressions. The judgement $\Delta \models P$ which appears in the typing rules for integer expressions denotes that under the context Δ, the (in)equality P holds. We leave a particular constraint solver up to implementation. It remains an open question whether undecidable constraints can be generated by our type system, though in practice it appears most common constraints can be efficiently solved with off-the-shelf constraint solvers [34].

The type system of Figure 9 also involves *kind* judgements of the form

$$\Delta \vdash \tau :: *$$

stating that τ is a simple type in the index context Δ. While the rules of our kind system are not given here, it is straightforward to derive. In particular, τ has kind $*$ if τ does not reference any free index variables, and does not contain any registers of negative length.

Remark 1. The fact that metaQASM has no means of specifying and checking relational properties on indices causes some programs to require counter-intuitive type schemes. For instance, the following n-bit adder is not well-typed due to the statement `toffoli(x[n-2], ctrl, y[n-1])`, though it does not cause run-time errors when $n \geq 2$.

Indices:

$$\frac{}{\Delta \vdash i : [i,i]} \qquad \frac{y : [I_1, I_2] \in \Delta}{\Delta \vdash y : [I_1, I_2]} \qquad \frac{\Delta \vdash I : [I_1, I_2] \quad \Delta \models I_1' \leq I_1 \quad \Delta \models I_2' \geq I_2}{\Delta \vdash I : [I_1', I_2']}$$

$$\frac{\Delta \vdash I : [I_1, I_2] \quad \Delta \vdash I' : [I_1', I_2']}{\Delta \vdash I + I' : [I_1 + I_1', I_2 + I_2']} \qquad \frac{\Delta \vdash I : [I_1, I_2] \quad \Delta \vdash I' : [I_1', I_2']}{\Delta \vdash I - I' : [I_1 - I_1', I_2 - I_2']}$$

$$\frac{\begin{array}{c} \Delta \vdash I : [I_1, I_2] \quad \Delta \vdash I' : [I_1', I_2'] \\ \Delta \models I_1'' = \min(I_1 \cdot I_1', I_1 \cdot I_2', I_2 \cdot I_1', I_2 \cdot I_2') \\ \Delta \models I_2'' = \max(I_1 \cdot I_1', I_1 \cdot I_2', I_2 \cdot I_1', I_2 \cdot I_2') \end{array}}{\Delta \vdash I \cdot I' : [I_1'', I_2'']}$$

Expressions:

$$\frac{\Delta; \Gamma \vdash x : \beta[I'] \qquad \Delta \models 0 \leq I < I'}{\Delta; \Gamma \vdash x[I] : \beta}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash E : \mathtt{Family}(y_1, \ldots, y_m)(\tau_1, \ldots, \tau_n) \\ \Delta \vdash I_1 : [0, \infty] \quad \cdots \quad \Delta \vdash I_m : [0, \infty] \end{array}}{\Delta; \Gamma \vdash \mathtt{instance}(I_1, \ldots, I_m) \ E : \mathtt{Circuit}(\tau_1\{I_1/y_1, \ldots, I_m/y_m\}, \ldots, \tau_n\{I_1/y_1, \ldots, I_m/y_m\})}$$

Unitary statements:

$$\frac{\Delta; \Gamma \vdash U : \mathtt{Unit}}{\Delta; \Gamma \vdash \mathtt{reverse} \ U : \mathtt{Unit}} \qquad \frac{\begin{array}{c} \Delta \vdash I : [I_1, I_2] \quad \Delta \vdash I' : [I_1', I_2'] \\ \Delta, y : [I, I']; \Gamma \vdash U : \mathtt{Unit} \end{array}}{\Delta; \Gamma \vdash \mathtt{for} \ y = I..I' \ \mathtt{do} \ \{ \ U \ \} : \mathtt{Unit}}$$

Commands:

$$\frac{\begin{array}{c} \Delta, y_1 : [0, \infty], \ldots, y_m : [0, \infty] \vdash \tau_1 :: * \quad \cdots \quad \Delta, y_1 : [0, \infty], \ldots, y_m : [0, \infty] \vdash \tau_n :: * \\ \Delta, y_1 : [0, \infty], \ldots, y_m : [0, \infty]; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash U : \mathtt{Unit}, \\ \Delta; \Gamma, x : \mathtt{Family}(y_1, \ldots, y_m)(\tau_1, \ldots, \tau_n) \vdash C : \mathtt{Unit} \end{array}}{\Delta; \Gamma \vdash \mathtt{family}(y_1, \ldots, y_m) \ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \ \{ \ U \ \} \ \mathtt{in} \ \{ \ C \ \} : \mathtt{Unit}}$$

Fig. 9. metaQASM typing rules

```
include "toffoli.qasm";
family(n) ctrlAdd(ctrl:Qbit, x:Qbit[n],
                  y:Qbit[n], c:Qbit) {
    toffoli(x[0], ctrl, y[0]);
    cx(x[0], c);
    toffoli(c, y[0], x[0]);
    for i=1..n-2 do {
        toffoli(x[i], ctrl, y[i]);
        cx(x[i-1], x[i]);
        toffoli(x[i-1], y[i], x[i])
    }
    toffoli(x[n-1], ctrl, y[n-1]);
    toffoli(x[n-2], ctrl, y[n-1]);
    for i=2..n-1 do {
        toffoli(x[n-i-1], y[n-i], x[n-i]);
        cx(x[n-i-1], x[n-i]);
        toffoli(x[n-i-1], ctrl, y[n-i])
    }
    toffoli(c, y[0], x[0]);
    cx(x[0], c);
    toffoli(c, ctrl, y[0])
}
```

The above adder can modified [7] to a well-typed program by using $m = n-2$ as the parameter, effectively specifying the number of entries *greater than* 2 that the input registers contain. The program snippet below gives the declaration required to make the controlled Adder implementation (with appropriate re-indexing) well-typed.

```
family(m) ctrlAdd(ctrl:Qbit, x:Qbit[m+2],
                  y:Qbit[m+2], c:Qbit)
```

In most practical cases appropriate parameters can be given so as to allow a well-typed implementation of a circuit family. However, the family parameters can be counter-intuitive, and more egregiously it can be unclear as to how to generate an intended instance. We leave it as an avenue for future work to add specification and checking of bounds and relational properties to metaQASM.

5.2   Type safety

As in the case of typedQASM, metaQASM is strongly normalizing, due to the lack of recursion and unbounded loops. Progress relies on the fact that during the course of evaluation, no free index variables are encountered – hence any term encountered by an interpreter is well-typed in the empty index context, and in particular indices can be evaluated to finite integers, as shown below.

Lemma 6. *If* $\cdot \vdash I : [I_1, I_2]$, *then* $\langle I, \sigma, \eta, |\psi\rangle\rangle \Downarrow i$.

Proof. Trivial since the judgement $\cdot \vdash I : [I_1, I_2]$ requires that I does not contain any variables. Note also that there is no derivation of a judgement of the form $\Delta \vdash \infty : [I_1, I_2]$ hence I cannot contain any infinite integers.

The remaining lemmas are extensions of results for typedQASM. Only the new or different cases are considered.

Lemma 7 (Preservation (expressions)). *If $\cdot; \Gamma \vdash E : \tau$, $\vdash \sigma : \Gamma$ and $\langle E, \sigma, \eta, |\psi\rangle \rangle \Downarrow v$, then either*
1. $\tau = \beta$ *and* $v = l$,
2. $\tau = \beta[I]$ *and* $v = (l_0, \ldots, l_{I'})$ *where* $I' \geq I$, *or*
3. $\tau = \mathtt{Circuit}(\tau_1, \ldots, \tau_n)$ *and* $v = \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U$
4. $\tau = \mathtt{Family}(y_1, \ldots, y_m)(\tau_1, \ldots, \tau_n)$ *and*

$$v = \Pi y_1, \ldots, y_m.\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U.$$

Proof. The new $\mathtt{Family}$ case is effectively identical to the $\mathtt{Circuit}$ case. For the case where $\tau = \beta$, it suffices to note that by Lemma 6, the expressions I and I' in the typing derivation reduce to integers i, i' and the proof concludes as in the typedQASM case.

Finally we have to revise the $\tau = \mathtt{Circuit}(\tau_1, \ldots, \tau_n)$ case as we now have two possible derivations. The new case $E = \mathtt{instance}(I_1, \ldots, I_m)\ E$ is also trivial as the only reduction produces a value of the form $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U$. Note that the type τ in the derivation has I_i substituted for index variables y_i, as in the conclusion of the reduction rule.

Lemma 8 (Progress (expressions)). *If $\cdot; \Gamma \vdash E : \tau$ and $\vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle E, \sigma, \eta, |\psi\rangle \rangle \Downarrow v$.*

Proof. Again, the new case $E = \mathtt{instance}(I_1, \ldots, I_m)\ E$ needs consideration. By inversion we see that $E : \mathtt{Family}(y_1, \ldots, y_m)(\tau_1, \ldots, \tau_n)$. By structural induction and the preservation lemma, $\langle E', \sigma, \eta, |\psi\rangle \rangle \Downarrow \Pi y_1, \ldots, y_m.\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.U$ and so $\langle E, \sigma, \eta, |\psi\rangle \rangle \Downarrow v$.

Lemma 9 (Progress (unitary stmts)). *If $\cdot; \Gamma \vdash U : \mathtt{Unit}$ and $\vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle U, \sigma, \eta, |\psi\rangle \rangle \Downarrow |\psi'\rangle$.*

Proof. The case $U = \mathtt{reverse}\ U$ requires a separate progress lemma for reverse reduction, which follows similar to progress for unitary statements.

For the remaining case $U = \mathtt{for}\ y = I_1..I_2\ \mathtt{do}\ \{\ U\ \}$, it suffices to observe that by inversion, $\cdot \vdash I_i : [I_i, I_i']$ and so both bounds reduce to integers. As each recursive call increases the lower bound I_1, and I_2 is necessarily finite, there can be no infinite chains of reductions. The only condition that needs checking is that $\langle U\{i_1/y\}, \sigma, \eta, |\psi\rangle \rangle \Downarrow |\psi'\rangle$, for which we need the following substitution lemma.

Lemma 10. *If $\Delta, y : [I_1, I_2']; \Gamma \vdash U : \mathtt{Unit}$, and $\Delta \vdash i_1 : [I_1, I_2']$ then $\Delta; \Gamma \vdash U\{i_1/y\} : \mathtt{Unit}$*

To complete the proof, another lemma is needed stating that the result of evaluating an integer expression is within the bounds of the expression's type. We leave this as an easy exercise.

Lemma 11 (Progress (commands)). *If $\cdot; \Gamma \vdash C : \texttt{Unit}$ and $\cdot; \cdot \vdash \sigma : \Gamma$, then for any $\eta, |\psi\rangle$, $\langle C, \sigma, \eta, |\psi\rangle\rangle \Downarrow \langle \sigma', \eta', |\psi'\rangle\rangle$.*

Proof. We have one new command to check,

$$C = \texttt{family}(y_1, \ldots, y_m) \ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \ \{ \ U \ \} \ \texttt{in} \ \{ \ C \ \}.$$

The proof in this case is effectively identical to regular gate declaration.

Theorem 2 (Strong normalization). *If $\cdot; \cdot \vdash C : \texttt{Unit}$, then*

$$\langle C, \emptyset, \lambda l.0, |00\cdots\rangle\rangle \Downarrow \langle \sigma, \eta, |\psi\rangle\rangle.$$

Proof. Follows directly from Lemma 11

6   Conclusion

We have described a typed extension to openQASM that supports static array bounds checking, higher-order circuits, and lightweight metaprogramming in the form of size-indexed families of circuits. The resulting language is powerful enough to use for writing libraries of general quantum circuit families, such as for reversible arithmetic, while low-level enough to be used wherever openQASM is used.

As this is preliminary work, much remains to be done to make metaQASM a practical language for quantum library development. In particular, a concrete implementation needs to be developed, as do more examples of practical circuit families. A major question which remains is whether a decision procedure for the simple, non-linear integer constraints generated by our type system exists.

Another interesting question for future work is whether *parametrized resource counts* for algorithms can be computed directly from metaQASM programs. In particular, a desirable feature would be to compute closed-form formulas for the number of qubits, gates, etc., in an arbitrary instance of a circuit family, so that different implementations of the same circuit family can be analytically compared *for any instance size*. Doing so would help not only with resource estimation, but also compilation by allowing compilers to automatically select the best implementation for a particular cost model.

Acknowledgements

References

1. Aharonov, D., Jones, V., Landau, Z.: A Polynomial Quantum Algorithm for Approximating the Jones Polynomial. In: Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing. pp. 427–436. STOC (2006). https://doi.org/10.1145/1132516.1132579
2. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: 20th Annual IEEE Symposium on Logic in Computer Science. pp. 249–258. LICS (2005). https://doi.org/10.1109/LICS.2005.1
3. Amy, M.: Feynman, `https://github.com/meamy/feynman`
4. Amy, M., Roetteler, M., Svore, K.M.: Verified Compilation of Space-Efficient Reversible Circuits. In: Proceedings of the 29th International Conference on Computer Aided Verification. pp. 3–21. CAV (2017). https://doi.org/10.1007/978-3-319-63390-9_1
5. Bello, L., Challenger, J., Cross, A., Faro, I., Gambetta, J., Gomez, J., Abhari, A.J., Martin, P., Moreda, D., Perez, J., Winston, E., Wood, C.: Qiskit, `https://github.com/Qiskit/qiskit-terra`
6. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open Quantum Assembly Language. arXiv preprint (2017), `https://arxiv.org/abs/1707.03429`
7. Fu, P.: Private communication (2018)
8. Gay, S.J.: Quantum Programming Languages: Survey and Bibliography. Mathematical. Structures in Comp. Sci. **16**(4), 581–600 (2006). https://doi.org/10.1017/S0960129506005378
9. Gheorghiu, V.: Quantum++: A modern C++ quantum computing library. PLOS ONE **13**(12), 1–27 (2018). https://doi.org/10.1371/journal.pone.0208073
10. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An Introduction to Quantum Programming in Quipper. In: Proceedings of the 5th International Conference on Reversible Computation. pp. 110–124 (2013). https://doi.org/10.1007/978-3-642-38986-3_10
11. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A Scalable Quantum Programming Language. In: Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 333–342. PLDI '13 (2013). https://doi.org/10.1145/2491956.2462177
12. Grover, L.K.: A Fast Quantum Mechanical Algorithm for Database Search. In: Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing. pp. 212–219. STOC (1996). https://doi.org/10.1145/237814.237866
13. Häner, T., Steiger, D.S.: 0.5 Petabyte Simulation of a 45-qubit Quantum Circuit. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 33:1–33:10. SC (2017). https://doi.org/10.1145/3126908.3126947
14. Häner, T., Steiger, D.S., Svore, K., Troyer, M.: A software methodology for compiling quantum programs. Quantum Science and Technology **3**(2), 020501 (2018). https://doi.org/10.1088/2058-9565/aaa5cc

15. Heyfron, L.E., Campbell, E.T.: An Efficient Quantum Compiler that Reduces T Count. Quantum Science and Technology **4**(1), 015004 (2018). https://doi.org/10.1088/2058-9565/aad604
16. JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: ScaffCC: Scalable Compilation and Analysis of Quantum Programs. Parallel Computing **45**(C), 2–17 (2015). https://doi.org/10.1016/j.parco.2014.12.001
17. Khammassi, N., Ashraf, I., Fu, X., Almudever, C.G., Bertels, K.: QX: A high-performance quantum computer simulation platform. In: Proceedings of the 20th Design, Automation Test in Europe Conference Exhibition. pp. 464–469. DATE (2017). https://doi.org/10.23919/DATE.2017.7927034
18. Khammassi, N., Guerreschi, G., Ashraf, I., Hogaboam, J.W., Almudever, C.G., Bertels, K.: cQASM v1.0: Towards a Common Quantum Assembly Language. arXiv preprint (2018), `https://arxiv.org/abs/1805.09607`
19. Killoran, N., Izaac, J., Quesada, N., Bergholm, V., Amy, M., Weedbrook, C.: Strawberry Fields: A Software Platform for Photonic Quantum Computing. Quantum **3**, 129 (2019). https://doi.org/10.22331/q-2019-03-11-129
20. Kissinger, A., van de Wetering, J.: PyZX: Large Scale Automated Diagrammatic Reasoning. arXiv preprint (2019), `https://arxiv.org/abs/1904.04735`
21. Kliuchnikov, V., Maslov, D., Mosca, M.: Fast and Efficient Exact Synthesis of Single-qubit Unitaries Generated by Clifford and T Gates. Quantum Information & Computation **13**(7-8), 607–630 (2013). https://doi.org/10.26421/QIC13.7-8
22. Liu, S., Wang, X., Zhou, L., Guan, J., Li, Y., He, Y., Duan, R., Ying, M.: $Q|SI\rangle$: A Quantum Programming Environment. arXiv preprint (2017), `https://arxiv.org/abs/1710.09500`
23. Lloyd, S.: Universal quantum simulators. Science **273**(5278), 1073–1078 (1996). https://doi.org/10.1126/science.273.5278.1073
24. Martonosi, M., Roetteler, M.: Next steps in quantum computing: Computer science's role. Computing Community Consortium (CCC) workshop report (2019), `http://arxiv.org/abs/1903.10541`
25. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge Series on Information and the Natural Sciences, Cambridge University Press (2000)
26. Ömer, B.: Quantum programming in QCL. Master's thesis, Technical University of Vienna (2000), `http://tph.tuwien.ac.at/~oemer/qcl.html`
27. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: A Core Language for Quantum Circuits. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 846–858. POPL (2017). https://doi.org/10.1145/3009837.3009894
28. Preskill, J.: Quantum Computing in the NISQ Era and Beyond. Quantum **2**, 79 (2018). https://doi.org/10.22331/q-2018-08-06-79
29. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. SFCS (1994). https://doi.org/10.1109/SFCS.1994.365700

30. Smith, R.S., Curtis, M.J., Zeng, W.J.: A Practical Quantum Instruction Set Architecture. arXiv preprint (2016), `https://arxiv.org/abs/1608.03355`

31. Soeken, M.: RevKit, `https://msoeken.github.io/revkit.html`

32. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: An Open Source Software Framework for Quantum Computing. Quantum **2**, 49 (2018). https://doi.org/10.22331/q-2018-01-31-49

33. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: *Q#*: Enabling Scalable Quantum Computing and Development with a High-level DSL. In: Proceedings of the 3rd ACM International Workshop on Real World Domain Specific Languages. pp. 7:1–7:10. RWDSL (2018). https://doi.org/10.1145/3183895.3183901

34. Xi, H.: Dependent types for program termination verification. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 231–242. LICS (2001). https://doi.org/10.1109/LICS.2001.932500