Verified translation between low-level quantum languages

Toward a verified compilation stack for Quantum Computing

KARTIK SINGHAL, University of Chicago, USA
ROBERT RAND, University of Maryland, USA
MICHAEL HICKS, University of Maryland, USA

We describe the ongoing development of a verified translator between OpenQASM (Open Quantum Assembly Language) and sQIR, a Small Quantum Intermediate Representation used for circuit optimization. Verified translation from and to OpenQASM will allow verified optimization of circuits written in a variety of tools and executed on real quantum computers. This translator is a step toward a verified compilation stack for quantum computing.

Additional Key Words and Phrases: Semantic Preservation, Formal Verification, Program Proof, Quantum Computing, Programming Languages, NISQ

1 INTRODUCTION

In the current Noisy Intermediate-Scale Quantum (NISQ) [Preskill 2018] era, we are limited by the availability of usable qubits on real machines and the errors that accumulate as circuit sizes grow. Hence, it is important to optimize circuits for low gate counts before running them on quantum computers.

VOQC [Hietala et al. 2019b] is a certified optimizer for quantum circuits written using the Coq proof assistant [Coq Development Team 2019]. VOQC's optimization functions are transformations of programs written in sQIR [Hietala et al. 2019a] (pronounced "squire"), a deeply embedded language that serves as VOQC's intermediate representation. These optimizations are proved correct using Coq.

While VOQC's optimizations are proved correct, not all of its passes have been. In particular, there is an initial pass that translates a circuit written in OpenQASM, the de facto standard circuit programming language, to sQIR, and this translation is unverified.

In this paper we present a translator between OpenQASM and sQIR and verify it using Coq. We prove a semantic preservation property between the sQIR's denotational semantics and a denotational semantics for OpenQASM based on Amy's [2019] big-step operational semantics. This represents a step toward a fully verified compilation stack for quantum computing.

2 SEMANTIC PRESERVATION BETWEEN QUANTUM PROGRAMMING LANGUAGES

In this section, we describe sQIR and OpenQASM, the translation between them, and the translation correctness property that we prove.

2.1 sQIR

sQIR or Small Quantum Intermediate Representation is a minimal low-level language deliberately designed for optimization of quantum circuits and implemented as a deep embedding in Coq. The syntax and denotational semantics of unitary portion of sQIR are shown below; Hietala et al. [2019a,b] provide complete details.

$$\begin{array}{rrcl} \text{Program} & P & ::= & P_1; P_2 \mid U\ q \mid U\ q_1\ q_2 \\ \text{Unitary} & U & ::= & \mathtt{H} \mid \mathtt{CNOT} \end{array}$$

$$\begin{array}{rcl} [\![P_1;\ P_2]\!]^{dim} & = & [\![P_2]\!]^{dim} \times [\![P_1]\!]^{dim} \\ [\![U\ q]\!]^{dim} & = & \mathrm{pad}_1(dim,\ U,\ q) \\ [\![U\ q_1\ q_2]\!]^{dim} & = & \mathrm{pad}_2(dim,\ U,\ q_1,\ q_2) \end{array}$$

Authors' addresses: Kartik Singhal, University of Chicago, USA, ks@cs.uchicago.edu; Robert Rand, University of Maryland, USA, rrand@cs.umd.edu; Michael Hicks, University of Maryland, USA, mwh@cs.umd.edu.

sQIR uses a global register of size *dim* to represent quantum state. Gates are then applied to the indices into that register. The denotation of gate application in sQIR is the unitary matrix produced after padding the gate matrix with identity operations (I) for all other qubits. For example:

$$\text{pad}_1(dim, \mathtt{H}, q) = (\mathtt{I}^{\otimes q} \otimes \mathtt{H} \otimes \mathtt{I}^{\otimes dim-q-1})$$

sQIR is parameterized by a set of unitary gates on one to three qubits. Above we only show a sample gate set of H and CNOT but in the formalization we use the standard Clifford+T set recognized by the VOQC optimizer. Whenever it is given a universal gate set, sQIR can express any finite quantum circuit.

2.2 OpenQASM

OpenQASM [Cross et al. 2017] is the most commonly used low-level quantum representation in practical use. Many major quantum programming languages and frameworks such as Q#, PyZX, Qiskit and Cirq target OpenQASM [Huisman 2018; LaRose 2019] as an intermediate representation to execute code on real machines [Wille et al. 2019].

We observe that at their core, OpenQASM and sQIR are very similar languages but differ in one key aspect. Namely, sQIR assumes a global, indexed register that makes it easy to refer to any qubit available to the program with its unique index; while OpenQASM uses abstract identifiers that need to be declared and looked up.

To verify correctness of a translation from/to OpenQASM, we must develop a denotational semantics for the language, which lacks one despite its pervasiveness. Amy [2019] provides an operational semantics for a subset of the language which we use as a starting point.

2.2.1 Syntax. We employ the following syntax of OpenQASM, following Amy [2019]:

Identifier	x		
Index	i		
Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\mathtt{H}(E) \mid \mathtt{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1;\ U_2$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

This syntax focuses on only the unitary fragment of OpenQASM. It also ignores gates parameterized by real numbers in favor of built-in gates. For example, instead of the unitary U gate that takes three real parameters, this syntax uses the Hadamard (H) gate from the OpenQASM standard library. This is a deliberate choice for the purpose of translation as most OpenQASM programs are written using the common gates defined in the standard library. Other single-qubit gates are similar modulo their matrix representation.

Even though we show a restricted syntax here, OpenQASM is clearly a much larger language than sQIR. Specifically, compared to sQIR, it requires declaration of qubit registers before being able to refer to qubits. Further, it allows user-defined unitary gates.

2.2.2 Semantics. To give a denotational semantics to OpenQASM, we also need the following semantic domains that represent values and are not part of the surface syntax:

$$\text{Loc},\ l \qquad\qquad \text{LocArray},\ (l_j, \ldots, l_k) \qquad\qquad \text{Gate},\ \lambda(x_1, \ldots, x_n).U$$

Locations correspond to individual qubits. LocArray similarly correspond to the locations that a register of qubit binds. User-defined gates correspond to functions. We define values, environments and quantum states with the following domain equations:

Value	V	=	Loc + LocArray + Gate
Environment	σ	=	Identifier $\rightarrow$ Value
Quantum State	$\lvert\psi\rangle$	=	Vector 2^m

Here $\lvert\psi\rangle$ is a vector in a 2^m-dimension complex Hilbert space, representing the complete quantum state available to a program. For a given program, the size of the vector is determined by $m = \sum n_i$ where n_i are the sizes of each of the declared quantum registers.

Similar to the semantics of sQIR, we need padding functions that are used to determine the modified quantum state after applying a unitary operation to the given location(s), for example:

$$\text{pad}(H, \lvert\psi\rangle, l) = (I^{\otimes l} \otimes H \otimes I^{\otimes m-l-1}) \lvert\psi\rangle$$

This manipulation of global quantum state is necessary as quantum computing inherently involves non-local effects such as entanglement.

We can now specify semantic functions for each of the three syntactic classes:

$$(\!| - |\!)_E : E \times \sigma \rightarrow V \qquad (\!| - |\!)_U : U \times \sigma \times \lvert\psi\rangle \rightarrow \lvert\psi'\rangle \qquad (\!| - |\!)_C : C \times \sigma \times \lvert\psi\rangle \rightarrow \sigma' \times \lvert\psi'\rangle$$

Expressions need an environment to return bound values. Unitary statements require an environment and the complete quantum state but only manipulate the quantum state. Finally, commands can manipulate both the environment (by declaring new registers or gates) and quantum states.

We elide formal details of denotations in this paper but they are essentially obtained by converting the big-step operational semantics for OpenQASM presented by Amy [2019] to a denotational style. More details will be made available in an upcoming full-length version of this paper.

2.3 Translation and Correctness

To translate from OpenQASM to sQIR, we maintain state that lets us map from a named register and index in OpenQASM into an index for the global register of sQIR. For the reverse direction, since sQIR does not have identifiers, we default to declaring a single qubit register while translating into OpenQASM. This way we get a clean translation using structural induction over the abstract syntax of both languages.

We want to show that the translations from sQIR to OpenQASM (which we will call f) and OpenQASM to sQIR (g) are semantics preserving:

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad [\![x]\!]^{dim} = (\!|f(x)|\!) \tag{1}$$

$$\forall y \in \mathcal{L}(\text{OpenQASM}), \quad (\!|y|\!) = [\![g(y)]\!]^{dim} \tag{2}$$

Both of these properties are easy to prove using structural induction on the abstract syntax. For (1), we create a single qubit register "q" of size m (dim of sQIR) to serve as the quantum state $\lvert\psi\rangle$ and add a single mapping for that in the environment σ. For (2), we also need to prove the correctness of the function that maps an OpenQASM register and its index to a sQIR global index.

Finally, we would also like to see that if we convert from one language to another and back, we obtain the identity function:

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad g(f(x)) = x \tag{3}$$

It is very easy to use our translation to obtain the original program written in sQIR and hence the composition of g and f is equivalent to identity function. But a similar statement for translating OpenQASM to sQIR and back does not hold. This is because of the lack of identifiers in sQIR. Our translation to sQIR is necessarily forgetful and hence, when we translate the program back to OpenQASM, we lose some of the structure of the original program.

We use ocamllex (the OCaml lexer generator) and the Menhir parser generator [Pottier and Régis-Gianas 2019] to write a feature-complete parser for OpenQASM. We then translate the unitary fragment of OpenQASM to that of sQIR. Our open source implementation and ongoing mechanization is available on GitHub.[1]

3 CONCLUSION AND PERSPECTIVES

Even though there exist verified quantum circuit generation languages like QWIRE [Paykin et al. 2017; Rand et al. 2017] and sQIR and the verified optimizing compiler VOQC, these are not yet widely adopted by practitioners. As we head into the age of quantum computing, it will be ideal to have a complete compilation stack that comes with mechanized proofs of correctness.

To bridge this gap, we present a verified translator between two low-level quantum languages, one used widely in the industry and another used for verified optimization in a research setting. In the process, we observe that the two languages line up well, but not perfectly and develop a denotational semantics for OpenQASM based on Amy [2019]. We prove a correctness property for our translation pertaining to this semantics and sQIR's denotational semantics.

Note that in this presentation we focused on the unitary fragment of both OpenQASM and sQIR. The next logical step is to also prove correctness of translation between the non-unitary fragments.

Additional steps toward a fully verified quantum compilation stack include validating our parser against the OpenQASM specification [Cross et al. 2017]. Jourdan et al. [2012] present a technique for validating LR(1) parsers which has been implemented as the Coq back-end for the Menhir parser generator. This back-end is used to generate the parser for the CompCert verified C compiler [Leroy 2009]. We believe that it should be fairly straightforward to rewrite our implementation that currently targets Menhir's OCaml backend into that for the Coq backend. Such a validated parser for OpenQASM would be another major component in the verified compilation toolchain for Quantum Computing. Another goal is to verify that a translation from QWIRE to sQIR is semantics preserving, allowing us to write programs in a safe higher-level language and safely compile them to sQIR or OpenQASM.

ACKNOWLEDGMENTS

This material is based upon work supported by EPiQC, an NSF Expedition in Computing, under Grant No. 1730449 and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Department of Energy.

REFERENCES

Matthew Amy. 2019. Sized Types for Low-Level Quantum Metaprogramming. In *Reversible Computation, RC 2019 (Lecture Notes in Computer Science)*, Michael Kirkedal Thomsen and Mathias Soeken (Eds.), Vol. 11497. Springer, Cham, 87–107. https://doi.org/10/gf8zzr

The Coq Development Team. 2019. The Coq Proof Assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303

Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019a. Verified Optimization in a Quantum Intermediate Representation. arXiv:1904.06319 Quantum Physics and Logic (QPL) 2019.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019b. A Verified Optimizer for Quantum Circuits. (July 2019). https://www.cs.umd.edu/~mwh/papers/voqc-draft.pdf

[1] https://github.com/inQWIRE/SQIR/tree/OpenQASM/qasm_to_sqir

Rolf Huisman. 2018. Q# Community Integrations. Retrieved Jun 22, 2019 from https://github.com/qsharp-community/qsharp-integrations "…one can run the quantum operations of a Q# application by using the OpenQASM output on the IBMQuantumExperience…".

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating *LR*(1) Parsers. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012 (Lecture Notes in Computer Science)*, Helmut Seidl (Ed.), Vol. 7211. Springer, Berlin, Heidelberg, 397–416. https://doi.org/10/gf9gsj

Ryan LaRose. 2019. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* 3 (Mar 2019), 130. https://doi.org/10/ggbnrq

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10/c9sb7q

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, Vol. 52. ACM, New York, NY, 846–858. https://doi.org/10/gf8t6s

François Pottier and Yann Régis-Gianas. 2019. The Menhir parser generator. http://gallium.inria.fr/~fpottier/menhir/

John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug 2018), 79. https://doi.org/10/gd3xfp

Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings of the 14th International Workshop on Quantum Physics and Logic, QPL 2017 (Electronic Proceedings in Theoretical Computer Science)*, Vol. 266. Open Publishing Association, Waterloo, NSW, Australia, 119–132. https://doi.org/10/gf8skv

Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, New York, NY, 1234–1240. https://doi.org/10/ggbj9p