Quantum Compilers Week 5: Let's build a compiler from scratch! CQ: A Small Classical-Quantum Hybrid Language

James Avery

Department of Computer Science (DIKU)
University of Copenhagen

Spring 2024 02196 Quantum Compilers Lecture Slides

Let's make a quantum language!

Design goals:

- D1: Simple: Easy to implement parser, interpreter, and compiler (we have 2.5 ECTS).
- D2: Convenient: Easy to work with for learning quantum program optimization techniques.
- D3: Useful: Easy to implement real useful quantum algorithms.

Design choices 1/4

Existing quantum algorithms are hand-tuned and written directly using a constant number of quantum gates.
It's desirable to design high-level quantum programming languages that lets us use higher abstractions, while still producing efficient quantum circuits.

But: This is an open research problem, violates D1. Get in touch if you're interested in exploring this with a thesis or a PUK!

Design choices 1/4

Existing quantum algorithms are hand-tuned and written directly using a constant number of quantum gates.
It's desirable to design high-level quantum programming languages that lets us use higher abstractions, while still producing efficient quantum circuits.

But: This is an open research problem, violates D1. Get in touch if you're interested in exploring this with a thesis or a PUK!

Design choices 2/4

Want: Quantum programs simple and tight, easy to manipulate and generate code for (D2).

Need: To program classes of quantum algorithms, we need a convenient way to program classical computer to generate tight quantum programs (D3).

D1+D2+D3: Separate classical and quantum operations.

Design choices 3/4: Choosing Programming Paradigm + Syntax

- D1 forces our hand: No fancy high-order functional or declarative paradigms.
- Let's choose to do an imperative language with a simple, familiar C-like syntax. Reduces mental overhead.

Design choices 3/4: Choosing Programming Paradigm + Syntax

- D1 forces our hand: No fancy high-order functional or declarative paradigms.
- Let's choose to do an imperative language with a simple, familiar C-like syntax. Reduces mental overhead.

Design choices 3/4: Choosing Programming Paradigm + Syntax

- D1 forces our hand: No fancy high-order functional or declarative paradigms.
- Let's choose to do an imperative language with a simple, familiar C-like syntax. Reduces mental overhead.

Design choices 4/4: Data Types

Which data types to support?

- No fancy composite datatypes (D1,D2).
  Also not needed for D3: common quantum algorithms only need qubits and constants.
- To program classes of quantum algorithms, we need integer computations on a classical computer, which controls the unitary operations to perform on the qubits. (D3)
- To do arbitrary rotations (D3: e.g. QFT, state input,...), we'll need "real numbers", in practice floating point numbers.
- Which classical data type should store results of quantum measurements? Could use integer type, but: If we use a separate data type, we can fully separate `ints` and `floats` from quantum operations $\rightsquigarrow$ introduce *classical bit* data type.

Design choices 4/4: Data Types

Which data types to support?

- No fancy composite datatypes (D1,D2).
  Also not needed for D3: common quantum algorithms only need qubits and constants.
- To program classes of quantum algorithms, we need integer computations on a classical computer, which controls the unitary operations to perform on the qubits. (D3)
- To do arbitrary rotations (D3: e.g. QFT, state input,...), we'll need "real numbers", in practice floating point numbers.
- Which classical data type should store results of quantum measurements? Could use integer type, but: If we use a separate data type, we can fully separate `ints` and `floats` from quantum operations $\rightsquigarrow$ introduce *classical bit* data type.

Design choices 4/4: Data Types

Which data types to support?

- No fancy composite datatypes (D1,D2).
  Also not needed for D3: common quantum algorithms only need qubits and constants.
- To program classes of quantum algorithms, we need integer computations on a classical computer, which controls the unitary operations to perform on the qubits. (D3)
- To do arbitrary rotations (D3: e.g. QFT, state input,...), we'll need "real numbers", in practice floating point numbers.
- Which classical data type should store results of quantum measurements? Could use integer type, but: If we use a separate data type, we can fully separate `ints` and `floats` from quantum operations $\rightsquigarrow$ introduce *classical bit* data type.

Design choices 4/4: Data Types

Which data types to support?

- No fancy composite datatypes (D1,D2).
  Also not needed for D3: common quantum algorithms only need qubits and constants.
- To program classes of quantum algorithms, we need integer computations on a classical computer, which controls the unitary operations to perform on the qubits. (D3)
- To do arbitrary rotations (D3: e.g. QFT, state input,...), we'll need "real numbers", in practice floating point numbers.
- Which classical data type should store results of quantum measurements? Could use integer type, but: If we use a separate data type, we can fully separate `ints` and `floats` from quantum operations ⇝ introduce *classical bit* data type.

Design choices 4/4: Data Types

Which data types to support?

- No fancy composite datatypes (D1,D2).
  Also not needed for D3: common quantum algorithms only need qubits and constants.
- To program classes of quantum algorithms, we need integer computations on a classical computer, which controls the unitary operations to perform on the qubits. (D3)
- To do arbitrary rotations (D3: e.g. QFT, state input,...), we'll need "real numbers", in practice floating point numbers.
- Which classical data type should store results of quantum measurements? Could use integer type, but: If we use a separate data type, we can fully separate `ints` and `floats` from quantum operations $\rightsquigarrow$ introduce *classical bit* data type.

Design Strategy 1: Separate Classical and quantum computations by types

Use

- three *classical* types: **int**, **float**, and **cbit**;
- and one quantum type, **qbit**.

Variables can either be

- *scalar* ("**qbit** x", one value) or
- *arrays* ("**qbit** x[*d*]", *d* values) of one of the four types.

Program constructs involving classical types are executed on a classical computer, and determine how the quantum controller programs the quantum computer, i.e., which quantum 'gates' will be executed for the qubits.

The classical- and quantum computer can interact only through **cbit** types: A **qbit** can be measured, resulting in the qubit collapsing to a single value, and the result being written to the **cbit**.

Design Strategy 1: Separate Classical and quantum computations by types

Use

- three *classical* types: **int**, **float**, and **cbit**;
- and one quantum type, **qbit**.

Variables can either be

- *scalar* ("**qbit** x", one value) or
- *arrays* ("**qbit** x[*d*]", *d* values) of one of the four types.

Program constructs involving classical types are executed on a classical computer, and determine how the quantum controller programs the quantum computer, i.e., which quantum 'gates' will be executed for the qubits.

The classical- and quantum computer can interact only through **cbit** types: A **qbit** can be measured, resulting in the qubit collapsing to a single value, and the result being written to the **cbit**.

Design Strategy 1: Separate Classical and quantum computations by types

Use

- three *classical* types: **int**, **float**, and **cbit**;
- and one quantum type, **qbit**.

Variables can either be

- *scalar* ("**qbit** x", one value) or
- *arrays* ("**qbit** x[*d*]", *d* values) of one of the four types.

Program constructs involving classical types are executed on a classical computer, and determine how the quantum controller programs the quantum computer, i.e., which quantum 'gates' will be executed for the qubits.

The classical- and quantum computer can interact only through **cbit** types: A **qbit** can be measured, resulting in the qubit collapsing to a single value, and the result being written to the **cbit**.

Design Synthesis 2: Pure quantum programs by partial evaluation

We construct the language to make it possible *evaluate away* all program constructs involving **int**s and **float**s, using a well-studied program transformation technique called *partial evaluation*.

$\rightsquigarrow$ generate a pure quantum program involving only **qbit**s,**cbit**s.

This pure quantum program will be a program in a simpler quantum-language subset to CQ, which we will call CQ^-.

We will use CQ^- as an *intermediate language* on which we can perform the program transformations needed to run on a physical quantum computer, and for reasoning about the quantum computations.

Design Synthesis 2: Pure quantum programs by partial evaluation

We construct the language to make it possible *evaluate away* all program constructs involving **int**s and **float**s, using a well-studied program transformation technique called *partial evaluation*.

$\rightsquigarrow$ generate a pure quantum program involving only **qbit**s,**cbit**s.

This pure quantum program will be a program in a simpler quantum-language subset to CQ, which we will call CQ$^-$.

We will use CQ$^-$ as an *intermediate language* on which we can perform the program transformations needed to run on a physical quantum computer, and for reasoning about the quantum computations.

Design Synthesis 2: Pure quantum programs by partial evaluation

We construct the language to make it possible *evaluate away* all program constructs involving **int**s and **float**s, using a well-studied program transformation technique called *partial evaluation*.

$\rightsquigarrow$ generate a pure quantum program involving only **qbit**s,**cbit**s.

This pure quantum program will be a program in a simpler quantum-language subset to CQ, which we will call CQ$^-$.

We will use CQ$^-$ as an *intermediate language* on which we can perform the program transformations needed to run on a physical quantum computer, and for reasoning about the quantum computations.

CQ Grammar 1: Program structure

We define a program as a sequence of *procedure declarations*, consisting of a procedure name, a (possibly empty) list of parameters, and a *statement* that designates what the procedure does.

$$\langle program \rangle ::= \langle procedure \rangle^{+}$$

$$\langle procedure \rangle ::= \mathrm{ID}\ \texttt{(}\ \langle parameter\_declarations \rangle\ \texttt{)}\ \langle statement \rangle$$

$$\begin{aligned} \langle parameter\_declaration \rangle ::= &\ \mathrm{TYPE\ ID} \\ |&\ \mathrm{TYPE\ ID}\ \texttt{[}\ (\mathrm{ID} \mid \mathrm{INT})\ \texttt{]} \end{aligned}$$

The first procedure in the list is the *program entry point* (like the function `main()` in C/C++), and its parameters define the input to the program.

CQ Grammar 2: Statements

Statements: can be *assignments* variable; unconditional and controlled *quantum updates*; a quantum *measurement* of a **qbit** storing the result in a **cbit**; a *procedure call*; a conditional ("if") statement; a while-loop; or a block of statements:

$$
\begin{array}{lll}
\langle statement \rangle & ::= & \langle lval \rangle \texttt{ = } \langle exp \rangle \texttt{;} \\
& | & \langle procedure\_call \rangle \texttt{ ;} \\
& | & \texttt{if (} \langle exp \rangle \texttt{) } \langle statement \rangle \texttt{ else } \langle statement \rangle \\
& | & \texttt{while (} \langle exp \rangle \texttt{) } \langle statement \rangle \\
& | & \langle block \rangle \\
& | & \langle qupdate \rangle \texttt{ ;} \\
& | & \langle qupdate \rangle \texttt{ if } \langle exp \rangle \texttt{ ;} \\
& | & \texttt{measure } \langle lval \rangle \texttt{ -> } \langle lval \rangle \texttt{ ;}
\end{array}
$$

$$
\langle procedure\_call \rangle ::= \texttt{call } \mathrm{ID} \texttt{ (} \langle lvals \rangle \texttt{)}
$$

CQ Grammar 3: Statements continued

A *block* is a sequence of variable declarations followed by a sequence of statements; and a variable declaration can be a scalar variable, a scalar variable with an initial value, or an array with an initial value list:

$$
\begin{aligned}
\langle block \rangle &::= \texttt{\{}\ \langle declarations \rangle\ \langle statements \rangle\ \texttt{\}}\ \langle declaration \rangle ::= \text{TYPE}\ \langle lval \rangle\ \texttt{;} \\
&\mid\ \text{TYPE ID}\ \texttt{=}\ \langle exp \rangle\ \texttt{;} \\
&\mid\ \text{TYPE ID}\ \texttt{[}\ \text{INT}\ \texttt{]}\ \texttt{=}\ \texttt{\{}\ \langle exps \rangle\ \texttt{\}}\ \texttt{;}
\end{aligned}
$$

$$
\begin{aligned}
\langle lval \rangle &::= \text{ID} \\
&\mid\ \text{ID}\ \texttt{[}\ \langle exp \rangle\ \texttt{]}
\end{aligned}
$$

The types and identifiers are straightforward:

$$\langle TYPE \rangle ::= \texttt{int} \mid \texttt{float} \mid \texttt{cbit} \mid \texttt{qbit}$$

$$\langle ID \rangle ::= \text{[a-zA-Z\_][a-zA-Z0-9\_]*}$$

Grammar of CQ 4: Expressions

CQ expressions are exactly the expressions from last week:

$$
\begin{aligned}
\langle exp \rangle ::= \; & \text{INT} \\
| \; & \text{FLOAT} \\
| \; & \text{NAMED\_CONSTANT} \\
| \; & \langle exp \rangle \; \text{BINOP} \; \langle exp \rangle \\
| \; & \langle lval \rangle \\
| \; & \text{UNOP} \; \langle exp \rangle \\
| \; & \text{BUILTIN\_FUN1} \; \text{`('} \; \langle exp \rangle \; \text{`)'} \\
| \; & \text{BUILTIN\_FUN2} \; \text{`('} \; \langle exp \rangle \; \text{`,'} \; \langle exp \rangle \; \text{`)'} \\
| \; & \text{`('} \; \langle exp \rangle \; \text{`)'}
\end{aligned}
$$

Simply import the symbols from your disambiguated grammar like so:

```
%import .expression (INT, FLOAT, BUILTIN_FUN1, BUILTIN_FUN2, NAMED_CONSTANT, PE,MD,AS,CMP, ID,  exp)
```

NB: You will need to modify the precedence of the '`ID:`' symbol to '`ID.-1:`' in `expression.lark`, so that keywords are matched first.

CQ Grammar 5: The lists

In the above, I've used the convention of writing e.g. '`exps`' to mean a list of '`exp`'s. This has to be defined in the grammar, and is done as follows:

$\langle \mathit{parameter\_declarations} \rangle ::= (\langle \mathit{parameter\_declaration} \rangle\ (\ ,\ \langle \mathit{parameter\_declaration} \rangle)^*)?$

$\langle \mathit{declarations} \rangle ::= \langle \mathit{declaration} \rangle^*$

$\langle \mathit{statements} \rangle ::= \langle \mathit{statement} \rangle^*$

$\langle \mathit{lvals} \rangle ::= \langle \mathit{lval} \rangle\ (\ ,\ \langle \mathit{lval} \rangle)^*$

$\langle \mathit{exps} \rangle ::= \langle \mathit{exp} \rangle\ (\ ,\ \langle \mathit{exp} \rangle)^*$

Full grammar and helper files:

That's it! A single PDF with the grammar in one piece, and helper files for your work, can be downloaded here:

`http://www.nbi.dk/~avery/QCC/CQ.pdf`	Grammar
`http://www.nbi.dk/~avery/QCC/CQ.lark`	Skeleton Lark file to get you started
`http://www.nbi.dk/~avery/QCC/helpers.py`	Various helper functions (Updated regula
`http://www.nbi.dk/~avery/QCC/show.py`	Recurses through full CQ program.
`http://www.nbi.dk/~avery/QCC/initialize.cq`	Test program.

Problem 2.1: Write a parser for CQ

Now that you're familiar with Lark, writing up the parser for CQ's grammar should be straightforward.

Downloads `CQ.pdf` and `CQ.lark` from the previous slide, and write up the grammar in Lark. Use your expression parser from last week, or download my reference version here:
`http://www.nbi.dk/~avery/QCC/expression.lark`.

Test it by parsing the test program `initialize.cq` and display the resulting AST using the `show.py` program.

Problem 2.2: Write a type checker for single-procedure CQ programs

Write a set of mutually recursive functions that type-checks a program CQ with only a single procedure, and no procedure calls. (Procedure calls are the most difficult: we will save them for next week). You can lift the recursive structure from `show.py` and use helper-functions from `helpers.py` (updated since last week, so re-download).

The type-checker should check (at least) the following rules:

1. Variables are declared in the current scope.
2. **qbit** variables are only used in quantum operations.
3. Expressions contain only scalar classical variables.
4. Assignments $lval = exp$ must ensure that $lval$ is a scalar or an array element that can contain the type of exp, i.e. $type(lval) \geq type(exp)$ according to the type hierarchy **cbit** $<$ **int** $<$ **float**. **qbit**s are not allowed in assignments. Use `max_type(t1,t2)` from `helpers.py`.

Start by making the type checker for expressions (which should return the expression type), then for statements (which should check