Artifact Overview

Introduction

This is the artifact for the paper "Compositional Quantum Control Flow with Efficient Compilation in Qunity". This artifact contains the code for the Qunity compiler and interpreter, as well as examples of Qunity code and scripts to run tests and benchmarks. This artifact supports the following claims made in the paper:

- We created the first working implementation of a Qunity compiler while introducing new control flow constructs and a metaprogramming layer.
- We used differential unit testing to verify the correctness of the circuits output by the compiler.
- The optimizations introduced in this work significantly improve the qubit and gate counts of the circuits generated by the compiler.

Hardware Dependencies

No specialized hardware is required.

Getting Started Guide

You should have Docker installed. Download the and unzip the provided file `54.zip`. Then, go to the `qunity` directory. Run the following command (note that the dollar signs throughout these instructions just indicate the start of the command and are not to be entered):

```
$ docker compose build
```

This should set up all necessary dependencies. It may take about 5 minutes to do this. Then, run the following:

```
$ docker run -it -v ./qasm_out:/qunity/qasm_out -v
./diagrams:/qunity/diagrams qunity:latest
```

The `-v` flags create a bind mount linking your host machine and the Docker container, so that you can directly see the compiled files and generated images in the `qasm_out` and `diagrams` directories on your host machine. If you are using Windows or otherwise if you don't see these files appearing on your host machine, you might need to use absolute paths instead of relative paths in the above command. For instance, if you put the `qunity` folder on your desktop, then the command would have to be

```
$ docker run -it -v C:/Users/user/Desktop/qunity/qasm_out:/qunity/qasm_out
-v C:/Users/user/Desktop/qunity/diagrams:/qunity/diagrams qunity:latest
```

This should start a Docker container. Now, to test if everything works correctly, run the following in the container:

```
$ ./qunity-compile examples/bit0.qunity --analyze
```

The expected output is the following:

```
Starting compiler
Preprocessing
Typechecking
Compiling to QASM
Postprocessing
.
Outputting to file
Compilation done
qasm_out/bit0.qasm

Starting analysis script
Importing libraries
Analyzing file qasm_out/bit0.qasm
Loading circuit
Drawing circuit
Diagram in diagrams/circuits/bit0.png
Transpiling circuit
Qubits: 1
Depth: 1
Gates: 1
Simulating circuit
Results in diagrams/sim_results/bit0_sim_results.png
```

If you get permission errors when running the above, try running the following in the container:

```
$ sudo chmod -R 777 .
```

You should now be able to see the compiled QASM file in `qasm_out/bit0.qasm`. You should also be able to see the generated circuit diagram in `diagrams/circuits` and `diagrams/sim_results`. All subsequent commands in the instructions should be run inside the Docker container.

Step by Step Instructions

Instructions for Reproducing Table 3

To reproduce the benchmark results from Table 3 and verify the circuit efficiency claim, you can simply run:

```
$ ./benchmarks
```

This script should take approximately 10 minutes to run. It will output the qubit and gate counts for the unoptimized and optimized Qunity compiler, as well as the reference Qiskit implementation when available. It will also draw and simulate the optimized Qunity and Qiskit circuits, and you can check that the generated results histograms look nearly identical (with minor differences due to random sampling), although these histograms are not very useful for some of the examples where the measurement outcomes are uniform over a large number of states.

Note that the evaluation of the unoptimized Grover's algorithm example should time out as the generated circuit is too large - the number of qubits can be confirmed by looking at the generated `qasm_out/grover_unoptimized.qasm` file. The order finding example will not have a Qiskit implementation and is not supported by the unoptimized compiler, and the list sum oracle example is also not supported by the unoptimized compiler.

Running the Tests

To run the tests, use the following:

```
$ ./run-tests
```

This should take approximately 1 minute to run. This script runs some basic unit tests, and also performs the differential unit testing mentioned in Section 7. The results of this can be seen in the tests labeled `compile_file_correctness`. These run the Qunity compiler on the listed files and simulate the resulting circuit, comparing the result to the output of the Qunity interpreter. For files that output circuits too large to simulate classically, the compiler is run without comparing the results, just to check that no exceptions are raised in the compilation process - these tests are labeled `compile_file_no_error`. All tests should say "passed".

Reusability Guide

This artifact provides tools to compile and run user-provided Qunity programs, ensuring its reusability. In this section, we describe the structure of the artifact and provide instructions for using these tools.

File Structure

The main parts of the code for the Qunity parser, preprocessor, typechecker, compiler, and interpreter are found in `lib`. The `bin` directory contains driver files for running the compiler and interpreter. The `examples` directory contains many examples of Qunity code, which are used for testing. The `qunitylib` directory contains the Qunity standard library, `stdlib.qunity`. The `test` directory contains files used for testing. The Bash and Python scripts in the outer directory can be used to more conveniently run the code and analyze, draw, and simulate the resulting circuits with Qiskit.

Running the Qunity Interpreter

To run a single program using the interpreter:

```
$ ./qunity-run <filename>
```

To start an interactive Qunity REPL:

```
$ ./qunity-interact
```

You can enter Qunity expressions or create definitions in the interpreter. Inputs should be terminated by a double semicolon: `;;`. You can exit the REPL by typing `%quit;;`. Here is an example of a REPL session:

```
$ ./qunity-interact
<qunity> $0;;
Expression type: Bit
Isometry: true
Pure semantics:
|0>

Mixed semantics:
|0><0|

Possible measurement outcomes:
Probability 1.000000: $0

<qunity> def @share : Bit -> Bit * Bit := lambda x -> (x, x) end;;
<qunity> @share($plus);;
Expression type: Bit * Bit
Isometry: true
Pure semantics:
 0.707+0.000i
 0.000+0.000i
 0.000+0.000i
 0.707+0.000i

Mixed semantics:
 0.500+0.000i  0.000+0.000i  0.000+0.000i  0.500+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
 0.500+0.000i  0.000+0.000i  0.000+0.000i  0.500+0.000i

Possible measurement outcomes:
Probability 0.500000: ($0, $0)
Probability 0.500000: ($1, $1)

<qunity> $ListEmpty{2, Bit} |> @list_append_const{2, Bit, $0} |>
@list_append_const{2, Bit, $plus};;
Expression type: List{2, Bit}
Isometry: false
Pure semantics:
None

Mixed semantics:
```

```
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.500+0.000i  0.500+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.500+0.000i  0.500+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i
 0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i  0.000+0.000i
0.000+0.000i  0.000+0.000i

Possible measurement outcomes:
Probability 0.500000: @ListCons{2, Bit}(($0, @ListCons{1, Bit}(($0,
$ListEmpty{0, Bit}))))
Probability 0.500000: @ListCons{2, Bit}(($0, @ListCons{1, Bit}(($1,
$ListEmpty{0, Bit}))))
```

The interpreter will output the type of the provided expression, whether or not it is considered an isometry by the typechecker, the computed semantics, and the possible measurement outcomes with their probabilities. The operator semantics is represented in matrix form, with pure semantics written in bra-ket notation if the matrix is internally represented as sparse. The first prompt given in the above example simply prepared a $|0\rangle$ state. The next two prompts defined a "share" operation and used it to prepare a Bell state by sharing a $|+\rangle$ state in the computational basis. The last prompt above demonstrated the use of Qunity's list data structure, appending $|0\rangle$ and $|+\rangle$ to an empty list of capacity 2: the measurement outcomes show that the second element is equally likely to be measured as $|0\rangle$ or $|1\rangle$.

Running the Qunity Compiler

To compile a single Qunity file into OpenQASM 3:

```
$ ./qunity-compile <filename> [-o <out_filename>] [--analyze] [--
unoptimized] [--nopost] [--img-format={png|jpeg|svg}]
```

If no output filename is specified, by default it goes in the `qasm_out` directory. If `--analyze` is used, a circuit diagram will be generated, and the circuit will be simulated using Qiskit. If `--unoptimized` is used, the old version of the compiler will be run (does not support features present in some of the examples). If `--nopost` is used, postprocessing optimizations are not applied. The `--img-format` option allows the user to specify the desired format of the generated circuit diagram and results histogram images, which can be `png` (default), `jpeg`, or `svg`.

For instance:

```
$ ./qunity-compile examples/grover.qunity --analyze
Starting compiler
```

```
Preprocessing
Typechecking
Compiling to QASM
Postprocessing
...
Outputting to file
Compilation done
qasm_out/grover.qasm

Starting analysis script
Importing libraries
Analyzing file qasm_out/grover.qasm
Loading circuit
Drawing circuit
Diagram in diagrams/circuits/grover.png
Transpiling circuit
Qubits: 7
Depth: 614
Gates: 760
Simulating circuit
Results in diagrams/sim_results/grover_sim_results.png
```

The corresponding files in the `diagrams` directory should then display the generated circuit diagram and the simulation results histogram. The resulting histogram should have a high frequency associated with the bit string `01100`, which is the "correct answer" for the oracle in this example, and a low frequency for all other bit strings.

> **Note on interpreting the outputs**: While in the above example, it is clear that the bit string `01100` corresponds to `($0, ($1, ($1, ($0, ($0, ())))))` in the Qunity source code, the encoding of Qunity types into bit strings when considering variant types may not be as obvious. If a type has two constructors taking types `'a` and `'b`, then the first bit (or qubit) in the encoding specifies which constructor is used, and the rest of the bits contain either a value of type `'a` or one of type `'b`. If one type is larger than the other, the unused bits must be zero in a valid encoding. Data types with multiple constructors are treated as a sum type of the first constructor with the remaining ones. So, for instance, when you compile and simulate `examples/equal_superpos_trit.qunity`, the results should show an equal frequency of `00`, `10`, and `11`, since these correspond to `$Nothing{Bit}`, `@Just{Bit}($0)`, and `@Just{Bit}($1)` respectively. There is no `01`, since that would be an invalid encoding for the type `Maybe{Bit}`. Similarly, for `examples/grover_with_lists.qunity`, the bit strings that have nonzero frequencies correspond to the lists `[]`, `[0]`, `[0, 0]`, `[0, 1]`, `[1]`, `[1, 0]`, and `[1, 1]`. Those with the higher frequency have an odd number of ones.

To compile all the example Qunity programs:

```
$ ./compile-all-examples [--analyze] [--unoptimized] [--nopost] [--img-
format={png|jpeg|svg}]
```

Note that while running `./compile-all-examples` should only take about 20 seconds, running it with `--analyze` to generate all diagrams and perform all simulations can take approximately 5 minutes. Several of the examples are explicitly skipped during this process because they take too long to simulate.