Hacq: A circuit description language for quantum circuits

Tsuyoshi Ito
NEC Laboratories America, Inc.

1 Introduction

Hacq[1] is a circuit description language for quantum circuits, which was originally developed for the research project "Tools to Optimize Resources in Quantum Engineering (TORQUE)" in the Quantum Computer Science (QCS) Program of the Intelligence Advanced Research Projects Activity (IARPA). Hacq allows a programmer to write structured quantum circuits concisely. Quantum circuits written in Hacq can be easily converted to a netlist in the standard gate set in the QC format used by QCViewer [QCV] if they are small enough. Moreover, reflecting the need in the IARPA QCS Program, even for huge quantum circuits whose netlist would be too large to store or process, Hacq produces the metric such as the number of gates, the width, and the depth of the circuits converted to the standard gate set.

Hacq is developed as an embedded language in the functional programming language Haskell [HHPW07], meaning that Hacq is in fact a library in Haskell and that a circuit description in Hacq is just a Haskell program which uses this library. (More precisely, Hacq is written in Haskell with some extensions to the language implemented in GHC [GHC], the de facto standard compiler of Haskell.) Because of this, a programmer can use all language constructs and libraries in Haskell at circuit generation time when describing quantum circuits. This is a powerful feature which allows concise description of complex quantum circuits.

Hacq provides a programmer with a set of useful building blocks to describe various quantum circuits in fairly simple manner. Some of these building blocks are subcircuits such as the implementation of an adder, but most of the building blocks provided by Hacq are *combinators*, or functions to construct a circuit from one or more simpler circuits. An example of combinators in Hacq is control, which takes a quantum circuit for a unitary U as an argument and returns a circuit for controlled-U. Hacq provides several most common combinators, and moreover, if the circuit contains a certain pattern which does not fit the combinators predefined in Hacq, a programmer can also define a new combinator to describe a complex circuit. This programming paradigm of using combinators to describe complex objects is well-established in the functional programming community. In particular, Hacq uses a common variation of the combinator approach called *monadic combinators* [Wad95].

The distribution of Hacq contains a circuit library generated by Single Qubit Circuit Toolkit (SQCT) [KMM, KMM13] to convert circuits which use single-qubit rotation gates with arbitrary angles to circuits which do not use them. SQCT is not needed just to use Hacq, but it is needed to regenerate the circuit library.

This documentation is far from complete. Users are encouraged to check examples in the "examples" folder.

2 Output

Hacq processes quantum circuits in two stages: circuit conversion and output generation.

[1] "Hacq" is pronounced as /hæk/ just like the English word "hack." It stands for "**Ha**skell **c**ombinators for **q**uantum circuits."

In the first stage, Hacq converts a circuit description to a quantum circuit in some restricted gate set. Usually this restricted gate set is the standard gate set: [2]

- One-qubit Pauli gates (X, Y, and Z).
- The Hadamard gate.
- The controlled-NOT (or CNOT) gate.
- The S gate ($\pi/4$-phase shift gates around the Z-axis) and its inverse.
- The T gate ($\pi/8$-phase shift gates around the Z-axis) and its inverse.

In the computational basis, these gates correspond to the following matrices:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad S^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}, \quad T^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{pmatrix}.$$

Output generation produces either the netlist in the QC format or the metric report (or both). When Hacq is instructed to report the metrics of a circuit, it reports the following:

- The number of Hadamard gates, the number of CNOT gates, the number of S gates and its inverse, the number of T gates and its inverse.
- The width of the circuit, i.e., the maximum number of qubits which have to be kept simultaneously.
- The T-depth of the circuit, i.e., the depth of the circuit considering only T gates and their inverses.
- The T-area of the circuit, i.e., the sum of the lifetimes of each qubit measured in T-depth.

3 Wires and circuits

In Hacq, a *wire* means a one-qubit register. A *circuit* in Hacq has the set X of input wires and the set Y of output wires, where $X \subseteq Y$, and always represents an isometry matrix from the Hilbert space corresponding to wires X to the Hilbert space corresponding to wires Y. A *unitary circuit* is a circuit with $X = Y$, in which case a circuit represents a unitary matrix on the Hilbert space corresponding to wires X.

A circuit is represented as a value of type `m a`, where `m` is an instance of `MonadQuantumBase w` for some type `w`, and `a` is a return type. The circuit itself is constructed as a side effect, and the return value does not have to have anything to do with the circuit. The type of a wire is `w` and it depends on the type of `m` via functional dependency.

`MonadQuantum` is a subclass of `MonadQuantumBase`, and a programmer is expected to use `MonadQuantum` or one of its subclasses to describe a circuit. We focus on `MonadQuantum` and its subclasses here.

[2] It is also possible to preserve Toffoli gates in the circuit as they are. This is useful for visualization of the structure of a small quantum circuit.

4 Basic gates

Here are some of the predefined circuits which correspond to basic gates:

- applyX w: A circuit which applies the NOT gate (aka the Pauli X gate) to wire w. Both input and output consist of the single wire w.
- applyY w: Similar, for the Pauli Y gate.
- applyZ b: Similar, for the Pauli Z gate. However, the target of the gate is specified as a *bit* instead of a wire. See below for the explanation of bits.
- applyH w: Similar, for the Hadamard gate.
- applyS b inv: Similar, for the S gate when inv is False or its inverse when inv is True.
- applyT b inv: Similar, for the T gate when inv is False or its inverse when inv is True.
- newWire: A circuit which allocates a new wire which is in an unknown state. It returns the wire. The input is the empty set, and the output is the new wire.
- ancilla: Similar to newWire, but it allocates a new wire which is initialized to state $|0\rangle$. (In fact, Hacq treats newWire and ancilla identically, and it does not take advantage of the fact that the wire returned by ancilla is initialized to $|0\rangle$. But the distinction might be useful. The name ancilla was not a good choice because wires allocated by it are not necessarily used as an ancillary space.)

A CNOT gate can be described by using function applyX with the combinator control explained in the next section.

Functions applyZ, applyS, and applyT take bits instead of wires as arguments. A *bit* is either a wire, the negation of a wire, constant $|0\rangle$, or constant $|1\rangle$. We make distinction between wires and bits because it does not always make sense to apply a circuit to a constant. For example, it is not possible to apply NOT to a constant, because the value of constant cannot be changed. On the other hand, it is possible to apply CNOT with $|1\rangle$ as control, and it just means NOT. Technically, it is well-defined to use a constant in place of an input wire of a circuit if the matrix corresponding to the circuit is block-diagonal with respect to this input wire. A bit is represented by type Bit w, where w is the type for a wire.

5 Basic combinators

- control b cir is the circuit which applies cir controlled by the bit b being $|1\rangle$. If b is a wire, it works as a control qubit. If b is a constant, it is just a generation-time conditional. When b is a wire, it must not be an output wire of circuit cir.
- parallel cir_1 cir_2 is the composition of two circuits cir_1 and cir_2, with an additional guarantee by the programmer that no wire is used as an output wire of both cir_1 and cir_2. Hacq chooses to use parallel composition when it knows parallel composition is possible from this guarantee. That is, it produces the parallel composition if there are no control wires, and it produces the sequential composition if there are any control wires. The return value is the pair of the return values of cir_1 and cir_2. In what follows, composition of two or more circuits with this assumption is called *optionally-parallel composition*.
- parallel_ cir_1 cir_2 is similar except that it discards the return values of cir_1 and cir_2.

- with_ $prepare$ $body$ is the sequential composition of circuit $prepare$, circuit $body$, and the inverse (more precisely, the adjoint) of circuit $prepare$. with $prepare$ $body$ is similar with the difference that now $body$ is a function which takes the return value of $prepare$ as an argument (just like (>>=)). The typical use of with is for allocating ancillary space in combination with ancilla:

Listing 1: Allocation of an ancillary wire using ancilla and with.

```
with ancilla $ \w ->
  -- w is now a new wire initialized to |0⟩.
  -- Do something with w and other wires, and bring w back to |0⟩ in the end.
```

  After this circuit, the wire allocated by ancilla is automatically deallocated. In order for this to produce a meaningful quantum circuit, all the wires allocated in $prepare$ part must be brought back to state $|0\rangle$ when they are deallocated.

- cir_1 >> cir_2 means the sequential composition of two circuits. cir_1 >>= cir_2 is the same with passing the result from cir_1. These are standard combinators in Haskell.

- invert cir is the inverse of a *unitary* circuit cir. Note that we cannot apply it to a non-unitary circuit, in particular not to newWire or ancilla.

6 Replication and iteration

- replicateQ n cir is the same as Seq.replicateA n cir. That is, it is the sequential composition of n copies of circuit cir, and returns a Seq of length n, whose ith element is the return value of the ith iteration. It is provided by Hacq because some of the interpretations allow a huge optimization; in these interpretations, the running time of replicateQ n cir is $O(\log n)$ instead of $O(n)$.

- replicateQ_ n cir is the same as replicateQ n cir except that it discards the result. This function could be implemented as a function of more general type Applicative m => Int -> m a -> m () (see replicateA_ in src/Util.hs), but it is defined as a method of MonadQuantumBase for the same reason as replicateQ.

- replicateParallelQ and replicateParallelQ_ are the optionally-parallel versions of replicateQ and replicateQ_, respectively.

- genericReplicateQ_ and genericReplicateParallelQ_ are the "generic" versions of replicateQ_ and replicateParallelQ_ which take an Integral instead of an Int as parameter n.

- mapM, mapM_, forM, and forM_ in standard Haskell module Control.Monad provide the loop construct which produces the sequential composition of many similar circuits. These functions work only with a list; if you use a Seq instead of a list, use the more general versions with the same name in standard Haskell modules Data.Traversable and Data.Foldable.

- mapParM, mapParM_, forParM, and forParM_ are the optionally-parallel versions of mapM, mapM_, forM, and forM_, respectively.

7 Convenience functions

- cnotWire w b: Equivalent to control b $ applyX w.

- cnotWires ws bs: The same as mapParM_ (uncurry cnotWire) $ zip ws bs, but it produces a more efficient circuit when used inside a control combinator.
- swapWire w_1 w_2: A circuit which applies the SWAP gate to two distinct wires w_1 and w_2. Implemented as with_ (cnotWire w_1 (bit w_2)) $ cnotWire w_2 (bit w_1).
- swapWires ws_1 ws_2: The same as mapParM_ (uncurry swapWire) $ zip ws_1 ws_2, but it produces a more efficient circuit when used inside a control combinator.
- newWires n is equivalent to replicateParallelQ n newWire, and ancillae n is equivalent to replicateParallelQ n ancilla, but they always produce the parallel composition (even if there are control wires).
- controls bs cir: Add all bits in bs as controls. For example, if bs is [b_1, b_2, b_3], then controls bs cir is equivalent to control b_1 $ control b_2 $ control b_3 cir.
- ifThenElse b cir_1 cir_2: The sequential composition of control b cir_1 and control (negateBit b) cir_2, but it produces a more efficient circuit when used inside a control combinator.

8 Some assumptions made by Hacq

The types of circuits used in Hacq do not represent all restrictions required for correct operations. Although it would be desirable to define the set of assumptions needed for correct operations of Hacq, such a set of assumptions is currently not available. Instead, here are some notable assumptions.

As explained above, combinators with and with_ expect that any wires allocated by the *prepare* part will be returned to the $|0\rangle$ state when they are deallocated. However, this assumption is not checked, and nothing prevents a programmer from writing the following code:

Listing 2: An incorrect use of combinator with.

```
bad :: MonadQuantum w m => m ()
bad =
    with ancilla $ \w ->
      applyH w
      -- Incorrect: The ancilla is not restored to |0⟩.
```

A somewhat related assumption is that the value (of type w) of an ancillary wire never escapes its scope. For example, the following code is incorrect (although the error is not reported):

Listing 3: Another incorrect use of combinator with.

```
bad :: MonadQuantum w m => m w
bad =
    with ancilla $ \w ->
      return w
      -- Incorrect: The ancillary wire escapes its scope.
```

9 Design decisions and future possibilities

Hacq was originally designed for the TORQUE project in the IARPA QCS Program. Part of the requirements in the IARPA QCS Program was to produce an estimate of the amount of resource to implement certain quantum

algorithms in a fault-tolerant manner, and Hacq was used to produce the number of gates and the width of the logical quantum circuits, which were further processed to produce the estimates at the fault-tolerant level. This has a few implications on the design of Hacq:

- Because some of the tasks in the IARPA QCS Program required huge quantum circuits, Hacq provides a way to report the resource requirement without generating all the gates in the circuit one by one.
- Because the circuits to describe often contained a large amount of classical computation which must be performed coherently, Hacq takes care of the ancilla management.

Hacq is developed as a library in Haskell. Another option considered was to develop it as a language in its own, but the expressiveness of Haskell turned out to be very useful in describing complex circuits.

The monadic style is chosen for convenience to describe circuits. However, this limits the possibility of static analysis of described circuits. If we try to make the library choose the most efficient circuit from several choices in the future, static analysis will be needed, and the monadic style may become problematic. Switching from monadic combinators to arrow-style combinators [SD96, Hug00] to allow static analysis of circuits is an option to investigate.

About license of Hacq

Acknowledgments

The author thanks Austin Fowler, Alex Parent, and Martin Roetteler for helpful discussions. The development of Hacq was supported in part by the Quantum Computer Science Program of the Intelligence Advanced Research Projects Activity.

References

[GHC] GHC Team. Glasgow Haskell Compiler (GHC). `http://www.haskell.org/ghc/`.

[HHPW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, pages 12–1–12–55, June 2007.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[KMM] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Single Qubit Circuit Toolkit. `https://code.google.com/p/sqct/`.

[KMM13] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates. arXiv:1206.5236v4 [quant-ph], February 2013.

[QCV] QCViewer Authors, Institute for Quantum Computing, University of Waterloo. QCViewer. `http://qcirc.iqc.uwaterloo.ca/index.php?n=Projects.QCViewer`.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming: Second International School, Tutorial Text*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207, August 1996.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52, May 1995.