Overview of Quipper

Dan Girshovich - PHYS 4481 Final Project

5/19/14

Intro

Quipper is a framework for designing and simulating quantum circuits on a classical computer. It is embedded in Haskell, a strongly typed and purely functional general-purpose programming language. Haskell's sophisticated type system has gives Quipper highly abstract and generalized primitives which make declaring high-level operations on circuits (like scaling input size, gate transformations, and simulations) relatively simple. This report briefly demonstrates each of these through an example *not* given in the Quipper documentation: characterizing an approximation to the standard Quantum Fourier Transform (QFT) circuit.

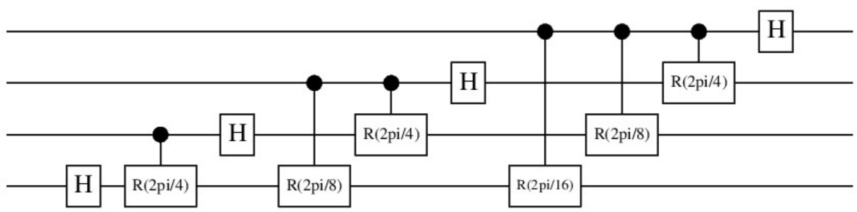
The Quantum Fourier Transform

The QFT circuit code is given as an example in the Quipper intrdductory documentation (slightly modified here):

```
qft :: QFunc
qft [] = return []
qft [x] = do
    hadamard x
    return [x]
qft (x:xs) = do
    xs' <- qft xs
    xs'' <- rotations x xs' (length xs')
    x' <- hadamard x
    return (x':xs'')
    where
        rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
        rotations _ [] _ = return []
        rotations c (q:qs) n = do
            qs' <- rotations c qs n
```

```
        let m = (n + 1) - length qs
        q' <- rGate m q `controlled` c
        return (q':qs')
```

These few lines encode the information to both simulate **and** draw the QFT circuit, using only primitives provided by Quipper and Haskell. Whats more, the number of input bits is abstracted over, so this code is general and reusable. Here is the circuit produced for $n = 4$ (note: the permutation step is ignored for this report):



This was generated by simply passing the `qft` function above to Quipper's `print_generic` high order function. Simulation is also a one-liner: passing the `qft` function and the equal superposition state $|\phi\rangle = \sqrt{\frac{1}{2^n}} \sum_x |x\rangle$ (encoded as a map from basis states to complex amplitudes) to Quipper's `sim_amps` function produces the expected delta function.

Approximating the QFT

As discussed in class and the textbook, a linear reduction in the number of gates used by the QFT is achieved by eliminating rotation gates below a certain threshold. In an ideal framework for designing quantum circuits, this approximation would be expressed as a transformation of the full circuit, not as a separate circuit. This way, the approximations themselves are first-class entities, which can be composed and abstracted over. Quipper provides this ideal setup through a more general construct called *circuit transformations*. Here is a circuit transformation I implemented to apply to "small phase error" QFT approximation:

```
-- removes all rotations smaller than 2 pi i / theta, where theta = 2 ^ (l + 1)
make_approx_transformer :: Int -> Transformer Circ Qubit Bit
make_approx_transformer l = transformer
    where
        transformer :: Transformer Circ Qubit Bit
        transformer (T_QRot name 1 0 inv theta ncf f) = f $
            \[q0] [] ctrls ->
```

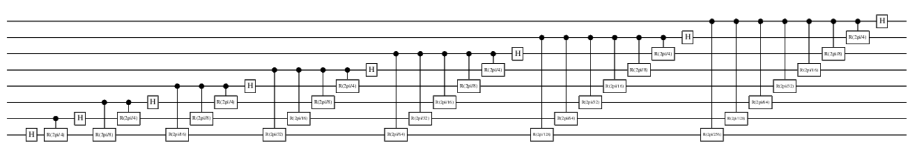
```
                without_controls_if ncf $
                with_controls ctrls $ do
                    unless (l < l') $ rGate_at l' q0
                    return ([q0], [], ctrls)
            where
                l' = (round $ logBase 2 theta) - 1
        -- leave everything else alone
        transformer g = identity_transformer g
```

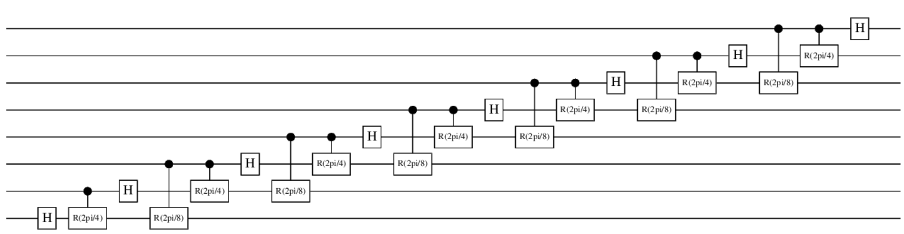
which is applied via:

```
make_approx_qft :: Int -> QFunc
make_approx_qft l = transform_generic (make_approx_transformer l) qft
```

As usual, l is the maximum distance between a rotation gate and it's control. Again, none of the functions used above are custom subroutines - everything is provided by Quipper and Haskell. Also, note the importance role that the type system plays in the circuit transformation. By pattern matching on `T_QRot`, the burden of only acting on **rot**ation gates is passed to the type checker. More importantly, the inner workings of Quipper's `transform_generic` function and `Transformer` type use highly generalized types to allow many different semantic maps to be applied to the circuit (for example, one that converts it to a classical simulation).

For comparison, here are the circuits for `qft` and `approx_qft` for $n = 8, l = 2$:





Characterizing the Approximated QFT

The circuit transformation allowed the implementation of an approximation parametrized over a strength parameter (l). An obvious use case for this setup is to characterize the error in the approximation given n and l. For more complex circuits with several levels of optimization and approximation, a feature like this may be the only feasible method for characterizing the error.

The following is a function I designed to perform several simulations of two circuits. For each simulation, a new random input state is generated and passed to both circuits. Finally, the errors in the outputs are accumulated and averaged.

```
-- simulate each circuit num_sims times and return the error rate
compare_circuits :: QFunc -> QFunc -> Int -> Double
compare_circuits c1 c2 num_sims = error_rate
    where
        error_rate = fromIntegral (length (filter id results)) / fromIntegral num_sims
        results = [meas g (sim c1 g) == meas g (sim c2 g) | s <- [1..num_sims], let g = mkSt
        sim circuit g = map snd . toAscList $ sim_amps g circuit $ input_state g
        input_state = make_state . normalize . take n . make_amplititudes . randoms
        make_state = fromList . zip basis_states
        basis_states = replicateM n [True, False]
        make_amplititudes (r1:r2:rs) = (Cplx r1 r2 :: Cplx Double) : make_amplititudes rs
```

The entirety of the code is included here for completeness:

```
{-# LANGUAGE  RankNTypes #-}

import Quipper
import Quipper.Monad (controlled)
import Quipper.QData (BType)
import QuipperLib.Simulation (sim_amps)
import Quantum.Synthesis.Ring (Cplx(Cplx))
import Control.Monad (unless, replicateM)
import System.Random (mkStdGen, randoms)
import Data.Map (Map, fromList, toAscList)
import Common


n = 8

l = 8


qft :: QFunc
qft [] = return []
qft [x] = do
    hadamard x
    return [x]
qft (x:xs) = do
```

```haskell
    xs' <- qft xs
    xs'' <- rotations x xs' (length xs')
    x' <- hadamard x
    return (x':xs'')
    where
        rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
        rotations _ [] _ = return []
        rotations c (q:qs) n = do
            qs' <- rotations c qs n
            let m = (n + 1) - length qs
            q' <- rGate m q `controlled` c
            return (q':qs')


-- removes all rotations smaller than 2 pi i / theta, where theta = 2 ^ (l + 1)
make_approx_transformer :: Int -> Transformer Circ Qubit Bit
make_approx_transformer l = transformer
    where
        transformer :: Transformer Circ Qubit Bit
        transformer (T_QRot name 1 0 inv theta ncf f) = f $
            \[q0] [] ctrls ->
                without_controls_if ncf $
                with_controls ctrls $ do
                    unless (l < l') $ rGate_at l' q0
                    return ([q0], [], ctrls)
            where
                l' = (round $ logBase 2 theta) - 1
        transformer g = identity_transformer g



make_approx_qft :: Int -> QFunc
make_approx_qft l = transform_generic (make_approx_transformer l) qft


approx_qft = make_approx_qft l


output_circuit_diagrams :: IO ()
output_circuit_diagrams = do
    output qft input_shape "qft"
    output approx_qft input_shape "approx_qft"
    where
        input_shape = replicate n qubit
```

```
-- simulate each circuit num_sims times and return the error rate
compare_circuits :: QFunc -> QFunc -> Int -> Double
compare_circuits c1 c2 num_sims = error_rate
    where
        error_rate = fromIntegral (length (filter id results)) / fromIntegral num_sims
        results = [meas g (sim c1 g) == meas g (sim c2 g) | s <- [1..num_sims], let g = mkSt
        sim circuit g = map snd . toAscList $ sim_amps g circuit $ input_state g
        input_state = make_state . normalize . take n . make_amplititudes . randoms
        make_state = fromList . zip basis_states
        basis_states = replicateM n [True, False]
        make_amplititudes (r1:r2:rs) = (Cplx r1 r2 :: Cplx Double) : make_amplititudes rs


main :: IO ()
main = do
    output_circuit_diagrams
    print $ compare_circuits qft approx_qft 3



-- Common.hs

module Common where

import Quipper (Circ, Qubit, print_generic, Format(EPS))
import Quipper.QData (qubit)
import Quantum.Synthesis.Ring (Cplx(Cplx), norm)
import System.Random (RandomGen, randomR)
import Data.Maybe (fromJust)
import Data.List (elemIndex)
import CatchStdOut (catchOutput)


type QFunc = [Qubit] -> Circ [Qubit]


output qf shape fname = do
    out <- catchOutput (print_generic EPS qf shape)
    writeFile (fname ++ ".eps") out


average :: [Double] -> Double
average ds = sum ds / (fromIntegral (length ds))


mag :: Cplx Double -> Double
```

```haskell
mag (Cplx r i) = sqrt (r**2 + i**2)


normalize :: [Cplx Double] -> [Cplx Double]
normalize as = [Cplx (ar * s) (ai * s) | (Cplx ar ai) <- as]
    where
        n = (fromIntegral . length $ as)
        s = (1/2) ** (n/2)


meas :: (RandomGen g) => g -> [Cplx Double] -> Int
meas g cs = chosen
    where
        amps = map ((**2) . mag) cs
        (r, _) = randomR (0, 1) g
        chosen = fst . last . takeWhile (\(_, v) -> v<r) . zip [0..] . scanl (+) 0 $ amps
```