Introduction to Quantum Programming and Semantics

Lecture 16: Oracles

Chris Heunen


University of Edinburgh

INFR11242 and INFR11243 Introduction to Quantum Programming and Semantics 2024/2025

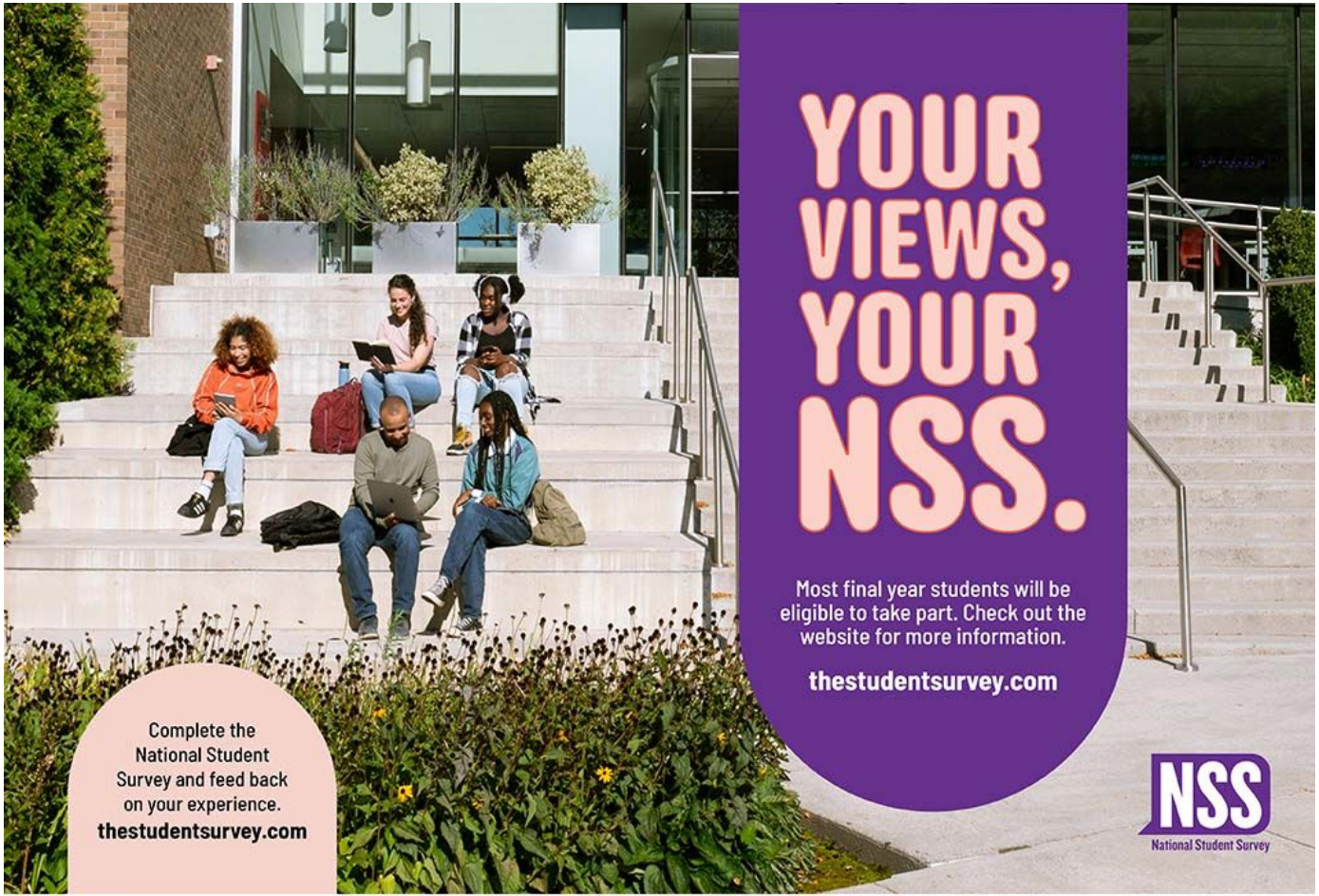
All students who fill in this form by the **Friday 25th April 2025** will be put in a draw to win one of two £25 vouchers.

Listening to and acting on student feedback is central to how we enhance the quality of our teaching. Responses will be anonymised before being shared with Course Organisers, along with Year Organisers and the Director of Learning and Teaching. These staff will use the responses to identify areas where we can improve courses and the programme next year.

Your responses will be anonymised, so that the staff mentioned above can only see a randomly generated ID. We will publish comments and how we plan to act on your feedback, but a small number of responses may not be sufficiently representative to draw particular conclusions, and so we will only publish the statistics if there is a sufficient response rate (10+).

We will remove responses that contain offensive language or otherwise breach the University Dignity and Respect policy (https://edin.ac/47Z4zAU).

Some questions in this survey are based on questions in the National Student Survey (NSS), conducted on behalf of the Office for Students (OfS). This is to allow us to compare responses to the NSS results.



Ipsos

Office for Students

Medr

Department for the Economy

Scottish Funding Council

Nominations are now open for the 2025 Teaching Awards!

Click the button below to nominate.

Edinburgh University Students' Association was the first in the UK to offer student-led Teaching Awards and our Awards are now in their 16th year. Staff play a pivotal role in the student experience at Edinburgh, from lecturers and tutors, to supervisors and personal tutors, and professional services staff. Our annual Teaching Awards provide students with an opportunity to thank staff for their hard work and celebrate the very best of teaching and support at the University, something that's just as important as ever.

Overview

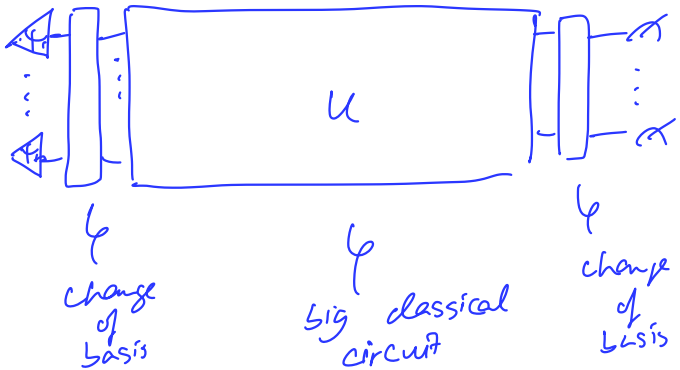
- Quipper
- Oracles

Types of quantum algorithms

main trick: change of basis

$-[H]-$, or more generally, quantum Fourier transform

① "oracular" quantum algorithms

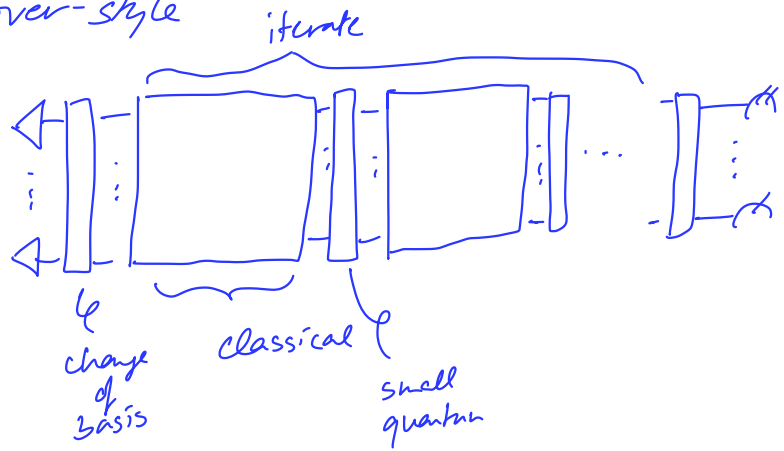


→ post-process

- Deutsch-Jozsa, Simon's
- Factoring (Shor): reduce factoring to period-finding
  $U(a) = a \bmod n$ + q: find n
- Hidden subgroup
  $i: G \to H$ injection of abelian groups
  find G

Types of quantum algorithms

② Grover-style

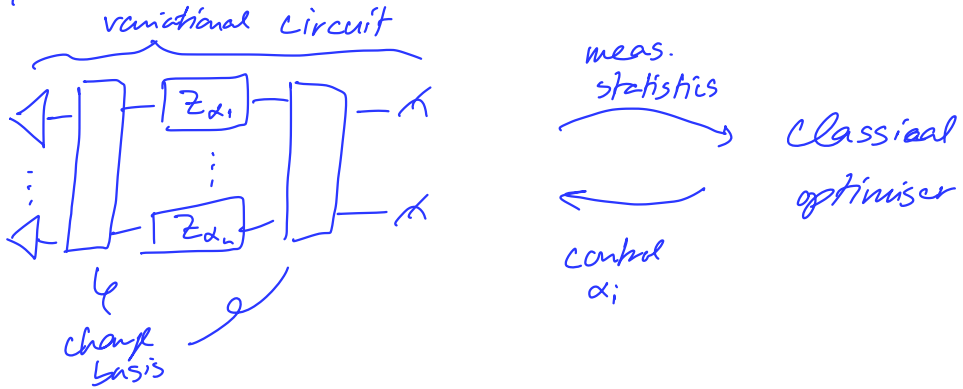


- Grover search
- amplitude amplification
- quantum walks

③ Hamiltonian simulation

Types of quantum algorithms

(4) Hybrid / quantum machine learning



Quipper

Quipper

- Open source
- Functional (with side effects)
- Domain-specific language in Haskell
- Aim: resource estimation
- Lazy
- Library including 7 nontrivial reference quantum algorithms

Quipper model

Execution phases:

- Compile time
- Circuit generation time
  Inputs whose values are already known now are called *parameters*
  e.g. Deutsch-Jozsa is really a family of circuits, one for each n
- Circuit execution time
  Inputs whose values are only known now are called *inputs*
  e.g. the n input qubits to Deutsch-Jozsa

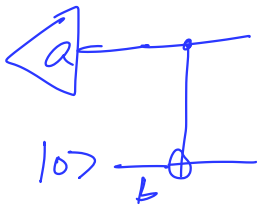
Quipper model

Types:

- Bits
  - have type Bool at circuit generation time
  - have type Bit as classical Boolean input to a circuit
- Qubits
  - Have type Qubit, only available as inputs at circuit execution time
- Bools can be converted into Bit, but not the other way around
- Measurements only at circuit execution time, so outcome is Bit, not Bool

Control

```
share :: Qubit -> Circ (Qubit, Qubit)
share a = do
  b <- qinit False
  b <- qnot b 'controlled' a
  return (a,b)
```



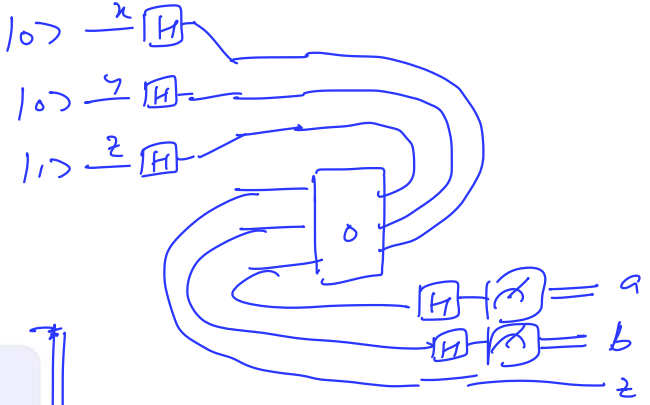
$|0\rangle \mapsto |00\rangle$

$|1\rangle \mapsto |11\rangle$

Measurement

```
measurement :: Qubit -> Circ Bit
measurement q = do
  x <- measure q
  return x
```

Quipper semantics



```
deutschjozsa :: (Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit))
                -> Circ (Bit, Bit)
deutschjozsa (oracle) = do
  x <- qinit False
  y <- qinit False
  z <- qinit True
  hadamard x
  hadamard y
  hadamard z
  (x,y,z) <- oracle x y z
  hadamard x
  hadamard y
  (a,b) <- measure (x,y)
  return (a,b)
```

=

Advanced Quipper

- Quantum data types:
  - E.g. (Qubit,[Qubit]), (Bit,[Bit]), (Bool,[Bool])

- Generic functions:
  - Following does not just apply to Bool, but also e.g. (Bool,[Bool])

```
plus_minus_generic a = do
  qs <- qinit a
  qs <- mapUnitary hadamard qs
  return qs
```

- Recursion:
  - Circuit-producing functions can be recursive over any parameters known at circuit generation time.
  - Can e.g. recurse over list of qubits to write QFT

Advanced Quipper

- Circuit operations:
  - Functions that take a circuit and make new circuits based on it
  - E.g. repeat circuit number of times, reverse circuit, use as subcircuit
  - Any classical reversible function can turn into a circuit automatically:

```
build_circuit
reversiblefunction :: (Bool, Bool, Bool) -> (Bool, Bool, Bool)
reversiblefunction x y z = y z x
```

- Semantics: well-founded semantics, but not yet
  - Dependent types: recognise that Bits are only used at circuit execution
  - Linear types: prevent terms involving Qubits from being copied

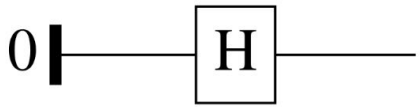
Proto-Quipper

Toy example

```
import Quipper

plus_minus :: Bool -> Circ Qubit
plus_minus b = do
  q <- qinit b
  r <- hadamard q
  return r

main = print_simple PDF (plus_minus False)
```



Deutsch-Josza

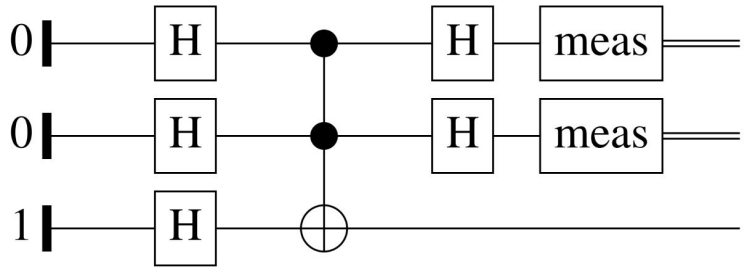
```
import Quipper

plus_minus :: Bool -> Circ Qubit
plus_minus b = do
  q <- qinit b
  r <- hadamard q
  return r

deutschjozsa :: (Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit))
                -> Circ (Bit, Bit)
deutschjozsa (oracle) = do
  x <- qinit False
  y <- qinit False
  z <- qinit True
  hadamard x
  hadamard y
  hadamard z
  (x,y,z) <- oracle x y z
  hadamard x
  hadamard y
  (a,b) <- measure (x,y)
  return (a,b)

oracle x y z = do
  qnot_at z `controlled` [x, y]
  return (x,y,z)

main = print_simple PDF (deutschjozsa oracle)
```



Summary:

- Quipper is functional, lazy, embedded
- Three phases of execution
- Higher-order so works particularly well with oracles