G53NSC and G54NSC
Non-Standard Computation

Dr. Alexander S. Green

26th of January 2010

Part I

Introduction

G53NSC and G54NSC - Non-Standard Computation

- Lecturer: Dr. Alexander S. Green (asg@cs.nott.ac.uk)
- Module Convener: Dr. Thorsten Altenkirch
- Module Webpage: http://www.cs.nott.ac.uk/~asg/NSC/
- Lectures: Tuesdays 11:00 to 13:00 (Business School South A24)
- Labs: Thursdays 15:00 to 17:00 (Computer Science A32)

What are the contents of this module?

- Non-Standard Computation...
- Any form of computation that doesn't follow the standard format of computation...
- What is computation?

What is Computation?

- What is computation?
- Computation is a general term for any type of information processing
- Computation is a process following a well-defined model that can be expressed as an algorithm
- What are algorithms?
- An algorithm is an effective method for solving a problem using a finite sequence of instructions

Alonzo Church
λ-calculus

Alan Turing
Turing machines

Church-Turing Thesis

Church-Turing thesis

Church-Turing thesis

All computational formalisms define the same set of computable functions

- What is meant by *all* computation formalisms?

Church-Turing thesis

Church-Turing thesis

All physically realisable computational formalisms define the same set of computable functions

- This thesis is believed by most people
- The subject area of Hypercomputing tries to challenge this.

What about complexity issues?

- We can write computable functions that take too long to actually compute in practise
- The best known algrithm for finding the prime factors of a large number is exponential in the size of the number to be factored
- However, primality testing (and multiplication), are only polynomial in the size of their arguments.
- The RSA encryption algorithm uses this anti-symmetry
- Current computers would take around a thousand years to break a 1024-bit RSA encryption key!

P versus NP

- The complexity class P contains computations that can be computed in polynomial time
- Computations in P are said to have efficient solutions.
- The complexity class NP contains computations that don't currently have efficient solutions. They are said to be unfeasible computations.
- It is still an unanswered question, but it is widely believed that $P \neq NP$
- Other complexity classes exist... (We shall look at a few later) For example, primality testing is in *BPP* *Bounded-error, Probabilistic, Polynomial time*
- Factorisation is currently in NP so isn't a feasible computation.

Extended Church-Turing thesis

Extended Church-Turing thesis

All physically realisable computational formalisms define the same set of feasible computable functions

- Non-Standard models of computation can challenge this
- What are these Non-Standard models of computation?

Non-Standard models of Computation

- DNA Computation is inspired by Molecular Biology
- Quantum Computation is inspired by Quantum Mechanics and Physics
- Cell Computation and P-Systems are inspired by Cell Biology
- This module will focus on *Quantum Computation*

Why Quantum Computation?

Peter Shor
Shor's Algorithm

- Shor discovered his probabilistic algorithm in 1994
- It can be used to factorise large numbers in polynomial time
- ... on a suitably sized Quantum Computer
- Quantum Computation seems to challenge the Extended Church-Turing thesis

How is this module evaluated?

- 50% Portfolio project
  consisting of weekly lab reports
- 50% Research report and presentation
  Individually for G54NSC students
  In pairs for G53NSC students
- with the possibility of a Viva...

Portfolio project

- Labs: Thursdays 15:00 to 17:00 (Computer Science A32)
- Exercises set weekly, using Haskell
  including work using the Quantum IO Monad, a library of functions for quantum computation in Haskell
- The last part of this lecture will be a Haskell refresher
- Overall deadline for portfolio: On course webpage
- Weekly Hand-ins suggested to enable continuous feedback

Research report and presentation

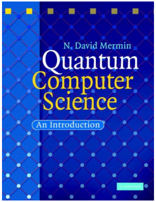
- Suggested topics available on course webpage
- Topic (and pairings for G53NSC) to be chosen by February 12th
- Each topic can only be done by one group (or individual for G54NSC)
- Get in early as topics are on a first-come first-serve basis
- After February 12th, pairings and topics will be allocated for you!

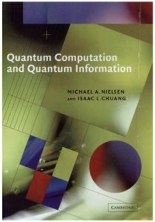
Research report and presentation

- Report in the form of a research paper on your chosen topic
- Presentations give an overview of the research paper
- Presentations are 12 minutes with 3 minutes for questions
- Presentations will be during the last two lectures
  Tuesday 23rd March, and Tuesday 30th March

Useful Material

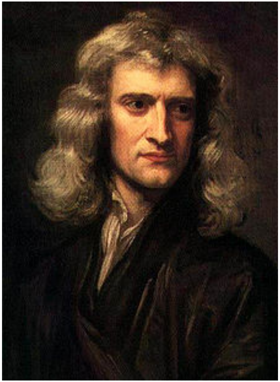
- The course website contains many useful links: http://www.cs.nott.ac.uk/~asg/NSC/
- The course will use the book: "Quantum Computer Science, An Introduction" by N. David Mermin (ISBN 0-521-87658-2)
- The book "Quantum Computation and Quantum Information" by Nielsen and Chuang is also very good (ISBN 0-521-63503-9)





Part II

A Brief introduction to Quantum Mechanics

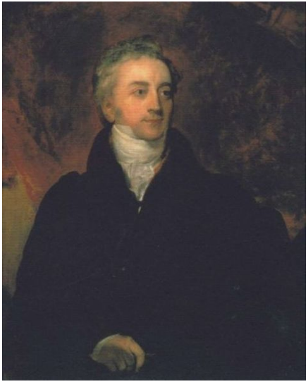


Isaac Newton
Light is made of particles

Christiaan Huygens
Light is a wave

Who is correct?

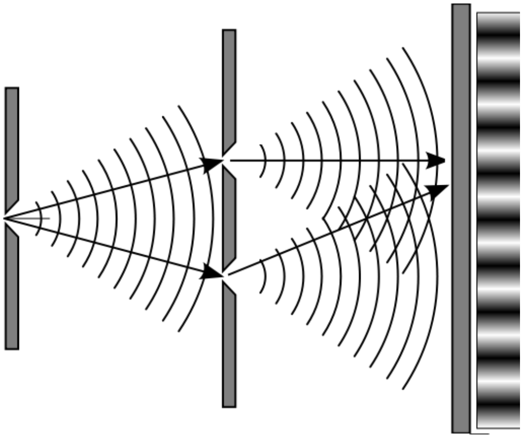
Young's Double Slit Experiment



Thomas Young
Young's double slit experiment

- The experiment involves shining light through two slits onto a screen
- If light is made of particles, we would see two bands of light
- If light is a wave, we would see an interference pattern
- What are we going to see?

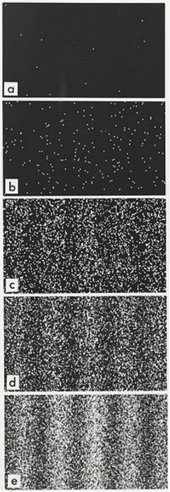
Young's Double Slit Experiment



- An interference pattern occurs
- Light is a wave?

Young's Double Slit Experiment

- But, what if we can slow this expermient down?
- Light now appears to arrive at the screen a single particle at a time
- Over time we still get an interference pattern
- Each photon must somehow interfere with itself



Wave-particle Duality

- At the *quantum* scale, matter exhibits both wave-like and particle-like behaviour
- E.g. Photons, and Electrons
- This is known as Wave-particle duality

The Copenhagen interpretation

Niels Bohr

Werner Heisenberg

Copenhagen interpretation of Quantum Mechanics

The Copenhagen interpretation

- The *state* of every particle is described by a wavefunction
- The wavefunction describes how a quantum state is a superposition of all possible classical states

The Born rule

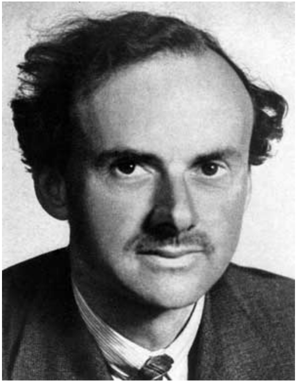
Max Born
The Born rule

- The probability of an event is related to the square of the amplitude of the wavefunction corresponding to it

The Copenhagen interpretation

- The *state* of every particle is described by a wavefunction
- The wavefunction describes how a quantum state is a superposition of all possible classical states
- The amplitudes correspond to the probability of observing a particle in a certain location
- Observation (or measurement) causes a wavefunction collapse, leaving the particle only in the state in which it was observed
- How can we talk about quantum states more formally?

Dirac notation



Paul Dirac
Dirac notation

- Dirac came up with the Bra-Ket notation for describing quantum states
- It is used extenisvely in the study of Quantum Mechanics and Quantum Computation
- Using Bras ($\langle\ |$) and Kets ($|\ \rangle$)

Dirac notation

- Kets ($| \ \rangle$) are used to denote the classical states in a quantum state
- with a corresponding complex valued amplitude
- We shall be using Dirac notation throughout this module...
- starting next week!
- What about the Labs this Thursday?

Labs on Thursday

- Lab exercises will make use of Haskell
- including advanced topics such as Monads
- We shall also be using the Quantum IO Monad, to write quantum computations within Haskell
- More information on the Quantum IO Monad is linked on the course webpage
- The rest of this lecture is a (re)introduction to the necessary Haskell for this weeks lab exercises

Part III

A brief (re)introduction to Haskell

Haskell

- Haskell is a functional programming language
- The functional paradigm means computations are defined in terms of function applications, and not variable assignments
- We will make use of the Glasgow Haskell Compiler's interactive system: GHCi
- GHC and GHCi are available online: http://www.haskell.org/ghc/
- The following slides are based on a similar lecture by Dr. Graham Hutton

Example

Summing the integers 1 to 10 in Java

```
total = 0;
for (i = 1; i <= 10; ++i)
   total = total + i ;
```

The computational method is variable assignment

Summing the integers 1 to 10 in Haskell

$$\mathit{sum}\ [1\mathinner{..}10]$$

The computation method is function application

Types in Haskell

- A type is a name for a collection of related values
- For example: the type

  Bool

- contains the two logical values:

  False
  True

Types in Haskell

- If evaluating an expression e would produce a value of type t, the e has type t, written

  $e :: t$

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called *type inference*

Types in Haskell

- Haskell has a number of basic types:
- *Bool* - logical values
- *Char* - single characters
- *String* - strings of character
- *Int* - fixed-precision integers

Lists in Haskell

- A list is a sequence of values of the same type

  $[False, True, False] :: [Bool]$
  $[\texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \texttt{'d'}] :: [Char]$

- In general, $[t]$ is the type of lists with elements of type t

Tuples in Haskell

- A tuple is a sequence of values of different types

  $(\mathit{False}, \mathit{True}) :: (\mathit{Bool}, \mathit{Bool})$
  $(\mathit{False}, \texttt{'a'}, \mathit{True}) :: (\mathit{Bool}, \mathit{Char}, \mathit{Bool})$

- In general, $(t1, t2, ..., tn)$ is the type of n-tuples with ith element of type ti for any i in $1..n$

Function Types

- A function is a mapping from values of one type to values of another type

  $not :: Bool \longrightarrow Bool$
  $isDigit :: Char \longrightarrow Bool$

- In general, $t1 \longrightarrow t2$ is the type of functions that map values of type $t1$ to values of type $t2$

Polymorphic Functions

- A functions is called polymorphic if its type contains one or more type variables

  $length :: [a] \rightarrow Int$

Pattern Matching

- Many functions have a particularly clear definition using pattern matching on their arguments

  $not :: Bool \rightarrow Bool$
  $not\ False = True$
  $not\ True = False$

Pattern Matching

- Functions on lists can be defined using $x : xs$ patterns

$$head :: [a] \rightarrow a$$
$$head\ (x : \_) = x$$

$$tail :: [a] \rightarrow [a]$$
$$tail\ (\_ : xs) = xs$$

Lambda Expressions

- A function can be constructed without giving it a name by using a lambda expression

  $\lambda x \rightarrow x + 1$

- Lambda expressions can be used to give a formal meaning to functions defined using currying

  $add\ x\ y = x + y$

means

$add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$

List comprehensions

- In Haskell, the comprehension notation can be used to construct new lists from old lists

  $[x^2 \mid x \leftarrow [1 \,..\, 5]]$

- The expression $x \leftarrow [1 \,..\, 5]$ is called a generator
- Comprehensions can have multiple generators

  $[(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$

gives

$[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$

Dependant Generators

- Later generators can depend on the variables that are introduced by earlier generators

$$[(x, y) \mid x \leftarrow [1 \mathinner{..} 3], y \leftarrow [x \mathinner{..} 3]]$$

gives

$$[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]$$

- Using a dependant generator we can define the library functions that concatenates a list of lists

$$concat :: [[a]] \rightarrow [a]$$
$$concat\ xss = [x \mid xs \leftarrow xss, x \leftarrow xs]$$

Guards

- List comprehensions can use guards to restrict the values produced by earlier generators

  $[x \mid x \leftarrow [1\,..\,10], \mathit{even}\ x]$

- Using a guard we can define a function that maps a positive integer to a list of its factors

  $\mathit{factors} :: \mathit{Int} \rightarrow [\mathit{Int}]$
  $\mathit{factors}\ n = [x \mid x \leftarrow [1\,..\,n], n\ \texttt{'mod'}\ x \equiv 0]$

Recursive functions

- In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive

$$\mathit{factorial}\ 0 = 1$$
$$\mathit{factorial}\ (n+1) = (n+1) * \mathit{factorial}\ n$$

For example, $\mathit{factorial}\ 3$

$$= 3 * \mathit{factorial}\ 2$$
$$= 3 * (2 * \mathit{factorial}\ 1)$$
$$= 3 * (2 * (1 * \mathit{factorial}\ 0))$$
$$= 3 * (2 * (1 * 1))$$
$$= 3 * (2 * 1)$$
$$= 3 * 2$$
$$= 6$$

Recursion

- Recursion is useful as properties of recursive functions can be proved using the mathematical technique of induction
- Recursion can also be used to define functions on lists

$$\begin{aligned}&\mathit{product} :: [\mathit{Int}] \rightarrow \mathit{Int}\\&\mathit{product}\ [\,] = 1\\&\mathit{product}\ (n : ns) = n * \mathit{product}\ ns\end{aligned}$$

Data Declarations

- A new type can be declared by specifying its set of values using a data declaration

  data *Bool* = *False* | *True*

- Values of new types can be used in the same ways as those of built in types
- In Haskell, new types can be recursive

  data *Nat* = *Zero* | *Suc Nat*

Type Constructors and Monads

- Haskell also allows us to define types that may contain other types

  data $Maybe\ t = Just\ t \mid Nothing$

- The first lab on Thursday will look at how we can use these Type Constructors to define Monads.
- Monads enable us to define impure computations within Haskell, which is a pure language
- We will be using the IO Monad to create a probabilistic primality test
- Later in the course we will be using the Quantum IO Monad to define quantum computations in Haskell
- Please see the course webpage on Thursday for more information