Quantum Gravity Seminar
Fall 2006

Classical versus Quantum Computation

28 Sept. 2006

Our first goal is to understand classical computation using the λ-calculus & Cartesian closed categories; later we'll try to:

1) quantize this (generalize to quantum computers)
2) categorify this (to see computations as a process: a 2-morphism)

Computability

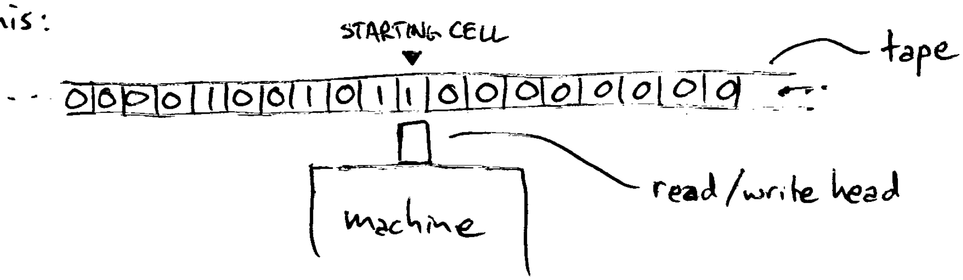
In 1936, Alan Turing tried to formalize the concept of a function

$$f : \mathbb{N} \longrightarrow \mathbb{N}$$

or partial function

$$f : \mathbb{N} \overset{\circ}{\longrightarrow} \mathbb{N}$$

(i.e. function defined on some subset $S \subseteq \mathbb{N}$) being "computable" by some repeatable process. He did this by inventing computers, or Turing machines. A Turing machine looks like this:



The machine has a finite set of states with distinguished <u>start</u> and <u>halt</u> states, & rules saying how at each step, the machine uses its current state & the bit on the cell the head is reading to:

1) write a 0 or 1 on that cell
2) move head left, right, or leave it fixed
3) change to a new state

If the machine reaches its halt state, it keeps writing the same bit, doesn't move, & stays in the halt state.

If we start with the binary number n on the tape & the machine halts at the starting cell with $f(n)$ written on the tape, $\forall n$, we say the machine computes f. If the machine doesn't halt at the starting cell, we say $f(n)$ is undefined. So Turing defined a notion of <u>computable</u> partial functions. He proved that (most) anything you can compute, a Turing machine can compute. But he also showed there are lots of uncomputable functions. Easy proof: there are countably many Turing machines, but uncountably many fns $f: \mathbb{N} \to \mathbb{N}$.

Better proof: he showed this function is uncomputable:

$$f(n) = \begin{cases} \text{undefined} & \text{if the } n\text{th Turing machine does halt when given the input } n \\ 1 & \text{otherwise} \end{cases}$$

He showed this is uncomputable by contradiction, using a diagonal argument.

When Turing submitted his paper for publication, he found Alonzo Church had scooped him, but using a different definition of computability, called <u>recursivity</u>. But his paper was published and he became Church's student. Church's definition used something called the "λ-calculus", which we'll discuss later. The two definitions were proved equivalent, along with many later ones. This suggested the

<u>Church-Turing Thesis</u>: any function $f: \mathbb{N} \to \mathbb{N}$ that can be computed by any repeatable process is computable in Turing's sense, & vice versa.

What's the λ-calculus? It's in some sense the simplest, most elegant framework for describing computable functions

$$f: \mathbb{N} \longrightarrow \mathbb{N}.$$

λ-calculus describes a world where every variable denotes a program (or "function") which can take as input other programs & output other programs! So what we have in this calculus are <u>λ-terms</u> (or <u>terms</u>) built from:

1) variables: $a, b, c, \ldots$

2) application: given a term X & a term Y, we have a term $X(Y)$ (heuristically: the result of using Y as input to X, or "applying" the fn X to Y)

3) abstraction: given any variable, say a, and a term X, we have a term

$$(a \longmapsto X)$$

(heuristically: the fn that maps a to X, typically something depending on a)

We're used to idea 3 combined with other ingredients, e.g.

$$x \longmapsto x^3 + 3x + 1$$

is a name for the fn $f: \mathbb{N} \to \mathbb{N}$ with $f(n) = n^3 + 3n + 1$ $\forall n \in \mathbb{N}$. But in the (simplest version of the "untyped") λ-calculus, the only ways of building terms are 1, 2, 3, so we get terms like

$$(x \mapsto (y \mapsto x(y))).$$

Let's apply this to some variable z and see what we get:

$$(x \mapsto (y \mapsto x(y)))(z) \underset{\substack{\uparrow \\ \text{this is called} \\ \text{"}\beta\text{-reduction"}}}{=} y \mapsto z(y) \underset{\substack{\uparrow \\ \text{this is called} \\ \text{"}\eta\text{-reduction"}}}{=} z$$

There's just one other rule in the λ-calculus besides β-reduction & η-reduction, and it's called "α-reduction": really just changing names of dummy variables, e.g.:

$$x \mapsto (y \mapsto x(y)) = q \mapsto (y \mapsto q(y))$$

Amazingly, from these rules we can build up Boolean logic, the natural numbers & their arithmetic operations, & ultimately use this to compute any Turing-computable function. We'll sketch this later.

In 1960, Landin showed the computer language ALGOL could be understood nicely using the λ-calculus.

In 1975, Steele & Sussmann invented SCHEME, a variant of LISP, explicitly based on the λ-calculus, including a primitive corr. to

$$(a \mapsto X)$$

which Church called "$\underline{\lambda\text{-abstraction}}$"

$$\lambda a. X$$

In 1980, Lambek realized the λ-calculus could be understood using <u>Cartesian closed categories</u>, which are categories that are (CCC)

1) <u>Cartesian</u>: they have finite products (it suffices to have binary products $a \times b$ and nullary products, i.e. terminal objects, written 1.

2) <u>closed</u>: for any objects x & y there is an object $\hom(x,y)$ or y^x s.t. there's a natural isomorphism between the set of morphisms

$$a \longrightarrow y^x$$

& set of morphisms

$$x \times a \longrightarrow y$$

for any object a.

The (untyped) λ-calculus is really all about a CCC with an object x s.t.

$$x \cong x^x$$

— explaining how anything is a program that eats programs & spits out programs.

For credit: do homework listed here:

http://math.ucr.edu/home/baez/qg-fall2006

(2 problems in Peter Selinger's notes)