Implementing quantum abstractions (Functional objects for quantum algorithms)

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen.

We show how to define *abstract* state vectors, their duals, and operators acting on them, as higher-order functions implemented in Haskell. Our model tries to “put into a computer” the standard quantum entities in a way as modest as possible, avoiding to invest more information than is permitted by the standard methodology of the quantum theory. In a sense we force a programmer to reason as a physicist. . .

Primary Message

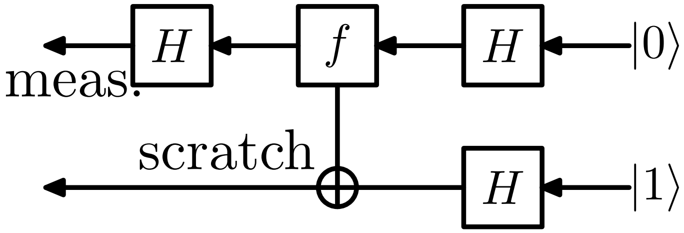
The simulation of quantum phenomena is difficult, since our epistemology, based on common sense and *senses* is classical; we **won’t ever feel** what is the quantum superposition of states.

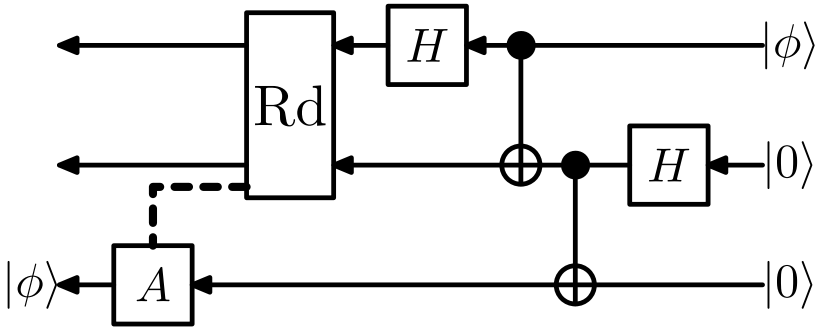
Yet, the idea of “quantum computing” is only partly due to the existence of (formally) ultra-fast algorithms, like Shor’s (factorization of integers) and Grover’s (search). Feynman noticed that a programmable quantum device will be useful for the simulation of other, complex quantum systems. No quanta without quanta, the complexity barrier is a killer…

But computer scientists need some classical intuition, and a bridge between what they learned from physics books, and their competence as implementors.

So, they speak about records and arrays representing quantum states, qubit vectors $\begin{pmatrix} p \\ q \end{pmatrix} = p\begin{pmatrix} 1 \\ 0 \end{pmatrix} + q\begin{pmatrix} 0 \\ 1 \end{pmatrix}$; about conditional statements and other decisional constructs, they play with the probabilistic reasoning (since quantum physics is non-deterministic), and they invent some “quantum computing languages” named QCL, QFT, etc., similar to classical ones. However: one of strengths of modern quantum theory is the recognition of the fact that states and observables are *physical entities* largely independent of the observer (of a concrete representation), despite the fact that for concrete computations some representation must be chosen. We shall thus try to **implement *abstractions***. “Quantum states” $|\psi\rangle$, etc. And we shall try to compute with them, as a physicist (or a student) does: formally, yet constructively, getting some numerical results.

We will show how to construct “qubit circuits” such as the “Deutsch oracle”, or a teleporting contraption:

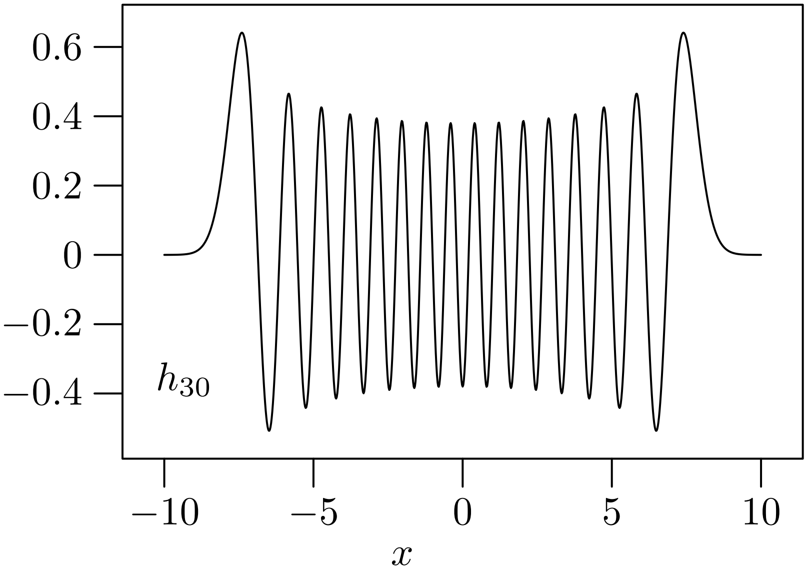




which are essentially compound (tensor) operators acting on quantum states, using *purely functional entities* to represent them. We will show also how to use functions to construct and to implement recursive formulae, permitting, say, to have a numerical recipe and the visualisation of the oscillator wave function $h_n(x) = \langle x|n\rangle$.

It fulfils

$$\langle x|n\rangle = \frac{1}{\sqrt{2}}\left(\sqrt{n}\langle x|n-1\rangle + \sqrt{(n+1)}\langle x|n+1\rangle\right)$$



and can be visualized as above almost directly from Haskell program (pumping the data into, say, Matlab).

Formal quantisation implemented functionally

The first stage is the construction of a quantum system. We know that a classical system, or its state is equivalent to a set of observable quantities. For example, a bit is a Boolean (or equivalent: 0 or 1). A particle has some position and momentum: $(\vec{x}, \vec{p})$. A spinning rotator has some angle, and the angular velocity, (φ, ω). (Or $\vec{\omega}$ if the axis is free.)

An oscillator can be treated as a particle: $(\vec{x}, \vec{p})$, but **we know** that after the quantisation its states can be numbered by a positive integer, the *discrete* excitation level n. The rotator needs two integers: j and $m = -j, -j+1, \ldots, j-1, j$ to enumerate the states. We know that.

The state of a quantum system is a vector in an abstract linear space equipped with a positive scalar product (metrics). The components of those vectors are **labelled with classical quantities**, so for a ***qubit*** we shall have two components, v_0 and v_1. The symbol v has no meaning, and we may follow Dirac: $|0\rangle$ and $|1\rangle$. The scalar product $\langle 0|\phi\rangle$ is the **probability amplitude** to obtain the result 0 while *measuring* $|\phi\rangle$. Any vector can be decomposed: $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$.

We shall introduce Haskell types which will permit to define **naturally** such abstract vectors, and their scalar products. We profit from the fact that the addition operator lifts naturally to the domain of functions: $(f+g)(x) = f\ x + g\ x$. Also: $c \cdot f(x) = c \cdot (f(x))$.

A general *classical* system may be modelled by a data structure describing some measurable quantities (outputs of experiments)

```
data Qubit = B0 | B1  deriving Eq -- here and later
data Osc   = X Double | P Double | N Integer
           | C Complex
data Rotator = Ang Double | JM Integer Integer
...
```

All those values constitute indices (or labels) of basic vectors in a metric (Hilbert) space: $(e_{\mathrm{B0}} \equiv |B0\rangle; e_{\mathrm{B1}} \equiv |B1\rangle$ etc. **Their interpretation is up to you!** We repeat that with p, q scalars, (usually complex), there exist objects like

$$|\psi\rangle = p \cdot |B0\rangle + q \cdot |B1\rangle.$$

We shall use the *Kolmogorov dilation theorem*, we postulate the existence of the “bracket” (or **kernel**) which represents the overlap between identical/different configurations. Its default value (the only classical one) is the Kronecker delta function, but there **are** also “non-orthogonal” configurations. Identical things yield 1, different — zero. This is the *restricted* orthogonality within the kernel.

```
class Eq a => Hbase a where
  bracket :: a -> a -> Scalar
  bracket j k = kdelta j k --(if j = k then 1 else 0)

instance Hbase Qubit -- etc.
--  sometimes overridden; no negative excitation
instance Hbase Osc where ...
  bracket (N j) (N k) | j>=0&&k>=0=kdelta j k
                      | otherwise = 0
```

A nice thing happens. Despite the fact that **`Hbase`** instances (the data structures) have no algebraic properties, a partially applied bracket (renamed: **`axis=bracket`**): **`axis alpha`** — for any **`alpha`**, **is a vector**, an element of a linear space, specified by the class **`Vspace`** containing the additive operations **`(<+>), (<->)`** and the multiplication by a scalar: **`(*>)`**.

The instances are constructed *naturally*:

```
type HV a = a -> Scalar
instance Vspace (HV a) where
  (f <+> g) x = f x + g x
  (f <-> g) x = f x - g x
  (c *> f)  x = c*(f x)
```

We shall also need operators, which transform vectors: **`HV a -> HV a`**, and multi-linear functions which correspond to *tensor states (compound systems)*. They will also belong to linear spaces through recursive instances for *n*-ary functions:

```
instance (Vspace a) => Vspace (b->a)
  (f <+> g) x = f x <+> g x
  (f <-> g) x = f x <-> g x
  (c *> f)  x = c*>(f x)
```

We can do linear arithmetic on functions, and compute, say, **`2*>axis (N 2)<->(1:+1)*>axis(N 0)`**. This is a finite object related to $2|2\rangle + (1 + i)|0\rangle$, belonging to an infinite-dimensional vector space.

How to make state vectors (Dirac kets)

What can we do with these (co-)vectors, what is their relevance to quantum physics and to computing?

We interpret **`(axis` α`)`** as a **co-vector** $\langle\alpha|$. We will need also the **dual** **ket** base which will give us $|\beta\rangle$'s, permitting to compute the amplitudes $\langle\alpha|\beta\rangle$, and we will be able to program *effectively* all standard (formal) calculations in text-book quantum mechanics. Plenty of results are obtained in an abstract setting, with vectors independent of concrete frames (bases).

Amplitudes need the existence of *linear functionals* over our vector space. Axes are vectors, but as functions they cannot be linear; the mathematical properties of **`Hbase`** are too weak. But the construction of formally linear functions is easier than you think!

An elementary “ket” corresponding to the value α, and dual to **`axis alpha`** is defined as

```
ket alpha = \ax -> ax alpha    --  ⟨ ax|α⟩
```

For any ket **`psi`** and axis **`phi`**, the construct: **`(psi phi)`** is the form $\langle\varphi|\psi\rangle$. Kets are vectors and *linear functions* over axes!:

```
ket a (ax1 + ax2)=(ax1 + ax2) a=ax1 a + ax2 a
                 = ket a ax1 + ket a ax2
```

The scalar product of two kets becomes also possible, since there is a *natural* duality operation: conversion of a ket into an axis, which implements: $\langle a|b\rangle = \overline{\langle b|a\rangle}$:

```
(dual kt) alpha = conjugate(kt (axis alpha))
```

You might appreciate the universality and beauty of the result, obtained without any specific datatypes representing vectors, and without imposing the linear structure by force.

Let’s repeat: functions are natural vectors. Functions which act on vectors (functions) by *calling them*, are naturally **linear functionals**. We got a Hilbert space for *any measurable* system “for free”.

And we might go further with this “bootstrap”, and out of $|\psi\rangle$ construct a bi-dual base, general “bras” $\langle\psi|$ **isomorphic to axes** through

```
(coax psi) chi = psi (dual chi)
```

again, because $\langle\psi|\chi\rangle = \overline{\langle\chi|\psi\rangle}$.

A bra can be lowered to an axis (proving the isomorphism). We need

`(lower br) alpha = br (ket alpha)  --` $\langle\, \text{br}|\alpha\rangle$

The following diagram is commutative.

$$\begin{tikzcd}
|\text{aKet}\rangle \arrow[r, "\textbf{dual}"] \arrow[dr, "\textbf{coax}"'] & \langle\text{anAxis}| \arrow[d, shift right, "\textbf{ket}"'] \\
& \langle\text{aBra}| \arrow[u, shift right, "\textbf{lower}"']
\end{tikzcd}$$

Functions acting on functions taking functional arguments? Are we mad?… Can it be used in practice? Yes, but we shall need *operators* acting on vectors. But first, two words about

Composite systems

To make a vector out of two others, we need to build-up their **tensor product**. We must dynamically construct functions with growing number of arguments in order to ensure the multi-linearity:

```
ax1<*>ax2=\alpha beta-> ax1 alpha*ax2 beta
k1<*>k2=\ax1 ax2-> kt1 ax1*kt2 ax2  -- $= |k_1\rangle|k_2\rangle$
```

etc. In general case we define the following class which takes into account that the result type is conditioned by the arguments:

```
class Tensor v1 v2 v3  | v1 v2 -> v3
 where
  (<*>) :: v1 -> v2 -> v3
```

The simplest instance of a tensor is a scalar. Others are specified recursively:

```
instance (Vspace v) => Tensor Scalar v v
 where
  s <*> v = s *> v
instance (Tensor v1 v2 v3)
 => Tensor (a->v1) v2 (a->v3)
  where
   u <*> v = \x -> u x <*> v
```

The construction of **entangled states**, e.g., $|\psi\rangle = (|01\rangle - |10\rangle)$ is straightforward:

```
psi = ket B0<*>ket B1 <-> ket B1<*>ket B0
```

Operators

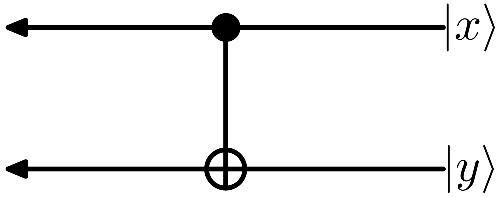
We need functions operating upon states. Some of them may implement the time evolution. Other are called **observables**, and represent measurable quantities. **All** experimental results are the "averages" $\langle\psi|\hat{A}|\psi\rangle$ of an observable $\hat{A}$ in the state $|\psi\rangle$. The probability $|\langle\alpha|\psi\rangle|^2 = \langle\psi|\alpha\rangle\langle\alpha|\psi\rangle$ is the average of the operator $|\alpha\rangle\langle\alpha|$, the projector on $|\alpha\rangle$. Its Haskell definition is **obvious from its visual form**:

```
(ktproj a) psi = psi(axis a)*>ket a -- = |α⟩⟨α|ψ⟩
```

It is easy to check that it is isomorphic to the tensor product

```
bra alpha <*> ket alpha   --  with
(bra alpha) kt = kt (axis alpha) --  = ⟨α| kt⟩
```

Other operators build from projectors or general “warps”: $|\alpha\rangle\langle\beta|$ are Haskell one-liners. The Hadamard operator $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ is defined as $1/\sqrt{2}(|0\rangle\langle 1| + |1\rangle\langle 0| + |0\rangle\langle 0| - |1\rangle\langle 1|)$, and its coding in Haskell is literal. The controlled-not gate which corresponds to the transition $|x\rangle|y\rangle \rightarrow |x\rangle|x \oplus y\rangle$ has a suggestive definition:



```
(cnot psi) ax ay = p B0 + p B1 where
  p b = psi (qproj b ax) (qxor (axis b) ay)
--  where
qproj b ax = ax b *> axis b  -- |b⟩⟨b|
qxor r = (r B0) *> id <+> (r B1) *> qnot
```

The generalized controlled-not, steered by a boolean function $f : |x\rangle|y\rangle \to |x\rangle|f(x) \oplus y\rangle$ is defined as

```
fcnot f psi ax ay = p B0 + p B1 where
  p b=psi (qproj b ax)(qxor (axis (f b)) ay)
```

And we may construct the Deutsch circuit. It is not so horrible. . .

```
warp alpha beta = \ax ->ax alpha *> axis beta
qnot = warp B0 B1 <+> warp B1 B0   --|0⟩⟨1| + |1⟩⟨0|
sigz = qproj B0 <-> qproj B1       --|0⟩⟨0| − |1⟩⟨1|
ahad = sqrt 0.5 *>(qnot <+> sigz)   -- Hadamard...
boost ax_op psi=\ax->psi(ax_op ax) --(⟨ ax|Â|ψ⟩)
boost2 aop1 aop2 = (boost aop2 .) . boost aop1
circuit f =
 let in1 = (boost2 ahad ahad) (ket B0<*>ket B1)
 in  boost2 ahad id (fcnot f in1)
```

The teleporting circuit has a comparable complexity, it remains a simple functional formula, with some general utilities (as **`warp`** or **`bost`** above), but we shall skip it, since it demands a thorough discussion of its interpretation…

Another example. In order to construct the "annihilation operator" **`ann`** for the base **`(N n)`** of the oscillator: $\hat{a}|n\rangle = \sqrt{n}|n-1\rangle$ (which is **an infinite sum** $\hat{a} = \sum_{n=0}^{\infty} \sqrt{n}|n-1\rangle\langle n|$), we define its "twin":

```
(ax_ann ax) (N n) = sqrt n * ax (N (n-1))
```

and we "lift" it: (**`ann=boost ax_ann`**)

We must here invest more knowledge, standard among physicists, that the position operator $\hat{x}$ is equal to $(\hat{a}^+ + \hat{a})/\sqrt{2}$, and a few other commonly known relations (e.g., defining the momentum $\hat{p}$ as well). We can then in three lines write an effective recursive algorithm which computes in Haskell the oscillator wave function shown before.

We won't show the details, since our objective is to suggest that the formalism is powerful enough to work with **infinitely-dimensional Hilbert spaces**, whose elements cannot be represented through standard data structures. . .

Conclusions and perspectives

We think that a decent functional language like Haskell is a reasonable tool for manufacturing an **abstract geometrical framework** with very non-trivial interpretation. This can be useful in quantum mechanics, but also for computations in differential geometry (thus: in almost all branches of theoretical physics. . .).

We can work effectively with **formal** entities; the translation between mathematical formulae and functional programs is short, straightforward and quite easy. We tried to show a particular case study which needs some math rarely taught to computer science students, and which requires some feeling of *real*, but not always numerical computational problems of the scientific community. Functional entities are universal and less constraining than “classical” datatypes.