Structure and Interpretation of Quantum Mechanics — a Functional Framework

Jerzy Karczmarczuk
Dept. of Computer Science, University of Caen
Caen, France
karczma@info.unicaen.fr

ABSTRACT

We present a framework for representing quantum entities in Haskell. States and operators are functional objects, and their semantics is defined — as far as possible — independently of the base in the Hilbert space. We construct effectively the tensor states for composed systems, and we present a toy model of quantum circuit toolbox. We conclude that functional languages are *right tools* for formal computations in quantum physics. The paper focuses mainly on the representation, not on computational problems.

Categories and Subject Descriptors

D.1.1 [**Programming techniques**]: Functional Programming

General Terms

Theory

Keywords

Haskell, Quantum physics, Vector spaces, Dual bases, Operators, Tensor products, Quantum gates, Multi-parametric classes.

1. INTRODUCTION TO QUANTIZATION

1.1 How to model quantum objects?

Computer scientists became interested in quantum computing mainly because of the possibility to accelerate the solution of algorithmically hard problems, see e.g. [1, 2, 3], also [4] and many others. But — as Feynman [5] remarked in 1982, — perhaps the most promising direction of evolution of programmable quantum systems is not the "algorithmics", but the simulation of *other* quantum structures. This is also advocated by Preskill [6], and worked upon by others [7, 8, 9, 10].

It is thus legitimate to ask how to represent properties of general quantum structures in a computer. As noted in [11], and elsewhere, we need a thorough abstraction layer upon physical details, in order to work on circuits and algorithms. There are attempts to introduce specific programming structures for the design of typical (imperative) languages [13, 14], making it easier to code the transformations acting upon the quantum registers. Bird and Mu [11] discovered the applicability of functional languages for writing such codes in a particularly compact and elegant way, and propose to use functional compositions and monadic chains to deal gracefully with the (nondeterministic) measures. Amr Sabry [12] goes further, and develops in Haskell a more complete functional framework for the simulation of quantum processing units and the observation of results. He also points out some difficulties arising from the application of typical programming languages to a non-classical domain. For a review of other attempts to simulate quantum structures, mainly collections of qubits manipulated imperatively see [15].

Typical representations of quantum entities are algebraic, in the classical sense. A quantum state is a vector belonging to a space, whose dimension is the number of discernible measurement results, e.g., a *qubit* which classically can be "up" or "down" can be represented as a linear superposition of $\binom{1}{0}$ and $\binom{0}{1}$. A multi-qubit is a Kronecker (tensorial) product of such matrices and needs compound data structures. For a harmonic oscillator it is known that a measurement can yield its excitation level — an integer between 0 and ∞. Here we see that lazy data structures might be useful. For the analysis of quantum algorithms qubits usually suffice, but some papers, e.g. [16], show that in order to assemble a physical multibit quantum gate, it is useful to couple the elements with quantum oscillators, objects beyond the qubit layer. In 1996 E. Knill observed [17] that a future quantum computer will certainly be a hybrid, with a classical part actively engaged in *preparing and interfacing* the quantum part (hardly a surprise for physicists performing experiments on quantum systems, but needing screwdrivers as well...). These meta-operations will transcend the elementary qubit abstraction layer.

Thus, our ambition is to propose a *general*, not restricted to qubits, implementable, functional abstraction layer for quantum entities, which would be effective even in the (observable sectors of an) infinite-dimensional Hilbert space, and which would correspond formally and intuitively to the formalism used in classical texts devoted to quantum physics, e.g., [18, 19]. Quantum states will be *functions*, and we shall use Haskell to code them. The title of the paper has been inspired by the book [20], whose authors underline the methodological usefulness of generic, functional structures for the computational representation of physical entities.[1]

Our framework achieves the following.

- We have a unified programming paradigm for different quantum systems.

[1]But the similarity to the title of the book of R.I.G. Hughes, *Structure and Intepretation of Quantum Mechanics*, Harvard University Press, is a pure coincidence. This book might be quite interesting for philosophically-oriented readers.

Haskell'03, August 28, 2003, Uppsala, Sweden.

- Functional objects are *opaque*, their internal structure is not observable, they are like the conceptual quantum states used by physicists. The only thing we can do with is to transform them by operators, and to project them on some basis.

- The level of abstraction is very high. Abstract quantum states can be manipulated independently of the observational framework which yields numbers, the probability amplitudes. These amplitudes are scalar products: $\langle\phi|\psi\rangle$ of state vectors, Dirac "kets" $|\psi\rangle$. We are encouraged to work on *universal* properties of such vectors (such as duality), independent of their concrete instantiation.

- The linear (vector) structure imposed on states arises *naturally* and universally.

Operators (state transformers, observables, etc.) will be functions acting upon functions, so the necessity of having a decent functional programming framework is obvious. An "abstract" vector — a geometric entity independent of the coordinate system which would specify its components, when implemented, is a concrete object, but it is not a data structure, manipulated, e.g., by pattern matching. We have no access to its internal structure, so we cannot duplicate it without decomposing it in a concrete basis.

The article is structured as follows: first we construct the quantum states for some simple systems, and we introduce a set of generic operators acting on them. Then, we show how to construct composite systems through tensor products, which in functional spaces is simple to define, but awkward to implement... Here the Haskell multiparametric classes with functional dependencies are very helpful. We say a few words about measurements. For concreteness we analyse some examples of operations on quantum oscillators, and — of course — on qubits. We construct some simple quantum circuits within our framework, and we implement a few simple-minded algorithms, for illustration only. Some general remarks conclude the paper.

1.2 Manufacturing physical systems

A classical system from the modelling perspective is a set of observable states. A flip-flop (a one-bit system) has two states, say, **Up** and **Down**, or **B0** and **B1** (suggesting Booleans rather than orientation). A particle has a position **x** and a momentum **p**. A 3D rotator has an angular momentum: two real numbers describing the rotation axis, and its azimuthal speed. But systems which have identical configuration spaces may be very different. A one-dimensional oscillator can be described by the position and the momentum of the moving point, exactly as a free particle. But the space topologies are different in both cases. Because of the energy conservation, for the oscillating point **x** and **p** are *bounded*, and this changes the mathematical structure of its Hilbert space, it possesses a *discrete* (Fock) basis, a free particle doesn't. (This is similar to the case of Fourier representation: if a function is defined on a finite support, it can be expressed through a discrete Fourier series, while an infinite support demands a continuous Fourier transform). This discrete basis for the oscillator, which corresponds to its excitation levels, will be used in some examples below, since the oscillator is the most important system in the whole quantum physics. Passing to some concrete descriptions we introduce the following:

A classical configuration, for example **B1**, or (θ, ϕ)) should be consider as a *label*, an "index" of a vector in a metric space. A quantum state is represented by such a vector (*cum grano salis*; the norm is fixed, and the global phase factor is unphysical). The classical state has no vector space properties attached to it, so we treat it as a "symbolic" description of the chosen basis, but *far* the meaning of the word "symbolic" in computer algebra packages.

Some classical description elements are superfluous in a constraining way. One cannot independently specify the position **and** the momentum, or the axis and the azimuthal speed of the rotator. They constitute *alternative sets of representative vectors*. At the present stage we don't need to speak about the Heisenberg uncertainty, just accept that we can represent a particle either through its momentum, or through its position, in the same sense as a spinning particle may be represented alternatively in different coordinate frames. Of course, conversions between those frames are possible.

Here the introduction stops. We are not reinventing Quantum Mechanics, we are just implementing it, in a universal, but minimalist way, using Haskell structures, so we *must* skip several justification steps. We can define thus some "physical" systems, e.g.

```
data Qubit   = Up | Down
--   You may ask wrt. which axis, but don't.[2]
data Mpoint  = Xc Double | Pc Double
--   Free particle
data Rotator = Ang Double Double
             | Jm Integer Integer
data Oscil   = X Double | P Double
             | N Integer
```

etc., where we observe:

- Each item defined at the right, represents a label ("index") set for the constructed vector space. Alternative bases are variants of these data structures. Since a particle can take an infinite number of positions, instead of enumerating them all, we use a parameterized data structure. We put together *all* classical (but conforming to quantum restrictions) configurations, all position vectors *or* all momentum vectors for a particle. Nb., for such non-denumerable cases functions will be obviously more natural than discrete data structures...

- A zero-dimensional set (a finite number) of index values implies a finitely-dimensional vector space, and a one-dimensional one, say, `X Double` specifies an infinite (here even non-denumerable) one. The `Qubit` datatype having two instances, is the foundation of a two-dimensional space with the basis vectors $\binom{1}{0}$ and $\binom{0}{1}$, or, in the Dirac notation: $|1\rangle$ and $|0\rangle$, or $|\uparrow\rangle$ and $|\downarrow\rangle$, etc. A basis state for a particle, denoted by $|x\rangle$ has a continuous set of components, for x in $\mathbb{R}^1$ (or $\mathbb{R}^3$, depending on the underlying geometry).

- For a rotator, the alternative to the **ang**ular dependency is a pair of integers (j, m) describing the "total" angular momentum, and its projection on *any* axis. A qubit might be implemented as a $j = 1/2, m = \pm 1/2$ rotator.

- In the oscillator example we introduced another (Fock) basis, the number `k` in `(N k)` is the level of energy, or the excitation number. Its typing is partly correct, the energy level k in $|k\rangle$ must be non-negative, while `Integer`s have no such restrictions, but we shall deal with such details in another way. This is the most frequently used basis for oscillators.

[2] The answer is ''ANY''. We cannot discuss the properties related to the underlying spatial substrate (if it exists; there are bi-level systems where the orientation does not play any significant role). In further examples we shall use labels **B0** and **B1**.

Depending on our needs, those bases may be augmented. For the oscillator, we may introduce another alternative basis, say, **`...| Ch Complex`**, a complex number which represent the amplitude of a so called *coherent state* — an "almost classical" wave packet base in which any compatible quantum state can be developed as well. They are very important for physics, begins to interest computer scientists, since logical qubits may be realized through coherent multi-photon states [21], but we cannot discuss them here.

Those data structures have rather weak mathematical properties, they are just labels related to, but not identified with the basic vectors tagged by them. Their parameters (numbers, Booleans, etc.) should be identifiable, which imposes that these data should belong to the class **`Eq`**, but more generally they should be *measurable*, since they correspond to classical, physical properties. In order to satisfy the superposition principle, we shall define now the relevant mathematics, making from our quantum states full-fledged vectors in a metric space, and we will also find a way to manufacture compound systems. The metric is fundamental, since the scalar products of state vectors give us the probabilities of measurements. Here the methodology of the paper diverges strongly from that of Amr Sabry [12] and others, who start with algebraic data items (tuples, lists or arrays) on which they impose the vector structure. We are going to ***construct*** this linear structure *ab ovo*.

2. BASIC PROGRAMMING WITH QUANTUM STATES

2.1 Induced vector structure

From the perspective of a functional programmer the problem consists in constructing some **linear** functions, say **`f`** such that, say, **`f (N 2)`** is the specified component of a *vector*. Although the specified data have no *a priori* arithmetic properties, we can easily give them to *functions* over those labels, by a known, standard construction, described in many books, e.g., [22]. We define some abstract addition and multiplication by a scalar as members of a class which represents vector spaces, and we say that some functions whose co-domain are Scalars, make the instance of this class:

```
infixl 7 *>
infixl 6 <+>,<->

class Vspace v where
 (<+>) :: v -> v -> v
 (<->) :: v -> v -> v
 (*>)  :: Scalar -> v -> v

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

where **`Scalar`** is usually a **`Complex Double`**, but other possibilities may also be interesting. Now we shall construct an induced metric. First we ***postulate*** the existence of a particular form, a "scalar product" for the state labels. We call this form the "bracket". For example, the form **`bracket (N j) (N k)`** is defined as a member of a particular type class:

```
class Eq a => Hbase a where
  bracket :: a -> a -> Scalar
  bracket j k = kdelta j k  --  Kronecker
instance Hbase Qubit
instance Hbase Oscil  --  etc.

kdelta a b = if a==b then 1 else 0
```

It corresponds to the physical requirement that different classical states are fully distinguishable, and it will generate the orthogonality properties of the true scalar product in the induced vector space usually denoted by the Dirac bracket $\langle j|k\rangle$. This holds only for discrete labels, in the continuous case we would need a more sophisticated apparatus: the generalized functions such as the Dirac delta. This is realizable, but cannot be discussed here.

The instances may override the default bracket, for example in the **`N`** base of the **`Oscil`** system the following holds:

```
bracket (N j) (N k)
        | j>=0 && k>=0 = kdelta j k
        | otherwise = 0
```

in order to eliminate the spurious negative levels. For the rotator discrete base: $|j,m\rangle$: $|m| \leq j$ must hold. There exist non-orthogonal bases as well (the coherent states for an oscillator is a good example thereof), and alternative bases have no reasons to be orthogonal, e.g., **`bracket (X x) (P p)`** is a complex exponential $\exp(ipx)$ which expresses the fact that a particle well localized in the momentum space is described by a plane wave (in this paper the Planck constant: $\hbar = 1$). In any case the brackets must fulfil the relation **`bracket a b = conjugate (bracket b a)`**, should be non-degenerate (not all vanishing), and positive: **`bracket a a`** is real, > 0.

For readability we introduce another name for **`bracket`**:

```
axis :: (Hbase a) => a -> HV a
axis = bracket
```

and *by postulate* the partially applied function **`axis`** α represents a ***basic adjoint*** state $\langle\alpha|$ for any α belonging to a **`Hbase`**. Axes are full-fledged vectors, we can write $(2+i)\langle 3| - 4\langle 1|$ as

```
f = (2:+1)*>axis(N 3) <-> 4*>axis(N 1)
```

etc.[3] We see here the power of a functional language, we have effectively created an "abstraction". The construct **`psi = axis (N 4)`** is opaque, we cannot extract its component, we can only check its value against another one, by applying it to, say, **`(N k)`**, and getting 0 or 1. This is a way the quantum elementary measuring processes are initiated, but this "filtering", and the construction of a probability amplitude needs in general also a "finalizing", feeding its square to a random number generator in order to get a concrete experimental answer).

A meaningful property of the structure imposed on the quantum states is that *physically* in the addition $\langle\chi| = \langle\phi| + \langle\psi|$ the two terms are evaluated in parallel, simultaneously, and the addition takes no physical time. In any classical simulation of quantum processes, this is impossible, and this distinguishes the complexity of quantum processes from their classical simulations.

In our implementation, in order to compute scalar products involving the combinations above we will need some *linear* functionals. Axes are auxiliary entities which cannot be linear because the **`Hbase`** has no associated algebra.

So, in the next step we define the dual base "ket"s: $|\uparrow\rangle$, $|n\rangle$, etc., as functions over our vector base (the axes). The term "vector" used generically will denote both axes and kets, but more specifically, the axes will be named *co-vectors*, in order not to forget the distinction between them. Again, Haskell permits to make a *universal* construction, the primitive kets, dual to elementary axes, are:

[3] **`a:+b`** denotes in Haskell the complex number $a + \mathbf{i}b$.

```
ket :: (Hbase a) => a->(HV (HV a))
ket alpha ax = ax alpha
```

(or **ket = flip id**, sometimes called the T combinator). The following test:

```
ax = 5.0*>axis(N 3) <-> 7.0*>axis(N 2)
kt = 9.0*>ket(N 2) <+> 2.0*>ket(N 3)
res = kt ax
```

gives **res**$= -53.0$, and the first stage of our construction is almost complete. Our abstract functional vectors have now sufficiently rich mathematical structure. Kets, combinations of **(ket** α**)** are functions belonging also to a **Vspace**, but, moreover, they are linear (the proof thereof is a useful exercise, showing how the linearity is "inherited").

One might raise a practical claim that it would be *easier* to represent **ax** as a lazy list: **ax=[0, 0, -7, 5, 0,...]**, etc., which would also permit to trivialize the duality operation. This would be dangerous, since scalar products are full reductors, not applicable directly to infinite lists. We couldn't compute the norm of a vector, while in our formulation finitely constructed functions yield always finite answers, unless badly used.

As mentioned above, axes are auxiliary vectors, physicists usually represent a state by a ket. We shall need thus the possibility to compute *scalar products* of kets: $\langle\phi|\psi\rangle$ of arbitrary $|\psi\rangle$ and $|\phi\rangle$, and in particular the squared norm $\||\psi\rangle\|^2 = \langle\psi|\psi\rangle$. Thus, we need duals to kets. They will also be useful for the construction of projection operators $|\psi\rangle\langle\psi|$. The dual to a ket **kt** should be an axis, a function over **Hbase**. The following should hold

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

We may simplify the notation even more:

```
dual :: (Hbase a) => (HV (HV a)) -> HV a
dual = conj . transp          -- where
conj f = conjugate . f
transp = boost axis
boost = flip (.)
```

Proof of the construction: if **kt = ket alpha** is an elementary ket, then

```
dual kt beta = dual (ket alpha) beta
 = conj (ket alpha . axis) beta
 = conjugate (ket alpha (axis beta))
 = axis alpha beta
```

which is correct. The linearity does the rest, **kt (dual kt)** yields 85. The construction seems unnecessarily complicated. For any **axes** objects $\langle\alpha|$ and $\langle\beta|$ we can compute $\langle\alpha|\beta\rangle$ as

$$\langle\alpha|\beta\rangle = \sum_\gamma \langle\alpha|\gamma\rangle\langle\gamma|\beta\rangle = \sum_\gamma \langle\alpha|\gamma\rangle\langle\beta|\gamma\rangle^* , \qquad (1)$$

where γ is a **Hbase** index. In fact, for a finite base (e.g., the qubits), this is an effective procedure. However if the base is infinite, but all the *concretely* constructed kets within the program come from finite linear combinations, our procedure yields the result after a finite number of steps, while the formal prescription (1) is ill-defined, and might never terminate. Moreover, the decomposition of a quantum state in a concrete basis from the physical point of view is not a neutral operation, it constitutes a measurement; formal insertion of $\mathbf{1} = \sum_\gamma |\gamma\rangle\langle\gamma|$ into a Dirac bracket is a purely formal trick, not done by Nature[4]. We shall use it in many scientific calculi, but it should be avoided — if possible — in the *simulation* of quantum circuits, apart from primitive gates, since it puts by hand a measurement inside a quantum process. And one of *raisons d'être* of our exercise is its methodological purity...

[4]This is a philosophical question: does Nature measure the components of unobserved state vectors? We don't think so...

The constructed framework gives a recipe for programming the quantum probability amplitudes for the state $|\psi\rangle$: $\langle\alpha|\psi\rangle$, or the physical measurement probabilities

$$P_\alpha(\psi) = |\langle\alpha|\psi\rangle|^2 \qquad (2)$$

that a system whose state is $|\psi\rangle$ yields upon a measurement the result which correspond to the component α (e.g., the spin is "down", or the oscillator finds itself at the ground level).

We may complete the **Vspace** class with the introduction of the zero vector, **vZero = const 0**. It is not needed as an independent object, but it is useful for the optimisation of some formulae.

2.2 General bras and bi-dual base

If we know how kets act on axes, we may reverse the problem, and find a bi-dual base of ***functions acting on kets, yielding scalars***. They are identified with arbitrary *bras* $\langle\ |$. (Recall that axes were functions over **Hbases** only, and we could not use them in arbitrary scalar products, although they spanned a vector space.) In order to transform a ket into a bra, we apply the function **coax**:

```
coax::(Hbase a)=>
  (HV (HV a)) -> HV (HV (HV a))
coax = flip id . dual
```

which expresses the identity **coax phi psi = psi (dual phi)** $\equiv \langle\phi|\psi\rangle$.

In particular, an elementary bra $\langle\alpha|$ belonging to this family, may be defined as **bra alpha = coax (ket alpha)**, or **bra = coax . ket**. This construction fulfills:

```
bra :: (Hbase a) => a -> HV (HV (HV a))
(bra alpha) kt = (kt (axis alpha))
```

and we may construct directly the linear combination of such bras without passing by the auxiliary axes.

The reader should observe that we have two transformations from the dual basis $|\ \rangle$ (kets) to $\langle\ |$ of two species: **dual** produces axes, while **coax** yields bras. The functional **ket = flip id** itself transforms axes into bras, the diagram on the Fig. 1 is commutative. (Thus, **ket** may be used in a more polymorphic context that it seems from its introductory definition.)

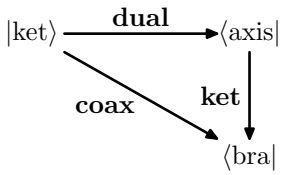

Figure 1: From kets to bras

On the other hand *there is no trivial (categorically universal) transformation from bras (including axes) to kets*, the only way is the decomposition of a bra in a basis, and the reconstruction of its dual from the coefficients. It involves thus a filtering, a part of measurement, and this is another part of our methodological defense against arguments that it would be much simpler to use standard matrices, where the construction of adjoints is conceptually, and technically easy. We refuse to "know too much".

Of course, the constructions above are not always required for actual solving of quantum problems, but show nicely some universal properties of the functional/geometric reasoning, and give us the feeling comparable to what we have manipulating abstract entities on paper. We show here how to develop an *implementable quantum formalism*, not how to construct quantum algorithms.

2.3 Operators

Linear operators: functions from vectors to vectors in quantum mechanics play primordial roles, some of them correspond to observables, other to symmetry transformers, and the time whole evolution of a quantum system is given by a linear operator. Actually, the only thing we can do with a quantum state, apart from computing scalar products, is to apply a linear operator to it. All quantum circuits are composition of linear operators. All observations involve the application of some "observable" operator. The operators form a vector space, the relevant class instance is

```
type HM b = HV b -> HV b
instance Vspace (HM b)
 where
  vZero v = vZero
  (f <+> g) a = f a <+> g a
  (f <-> g) a = f a <-> g a
  (c *> f)  a = c*>(f a)
```

and if the base is finite, they may be effectively represented as matrices. In functional representation the multiplication of operators is just their composition.

We may specify operators by their action on the **Hbase** objects, e.g., saying that a spin is inverted, or that **(N k)** should become **(N (k-1))**, etc., and lifting them to functions. But the construction of functionals acting on functional objects in this way, is delicate, we shall not forget that lifting the set X of objects to a vector space Fun(X) of functions on them is a *contravariant functor*. If we have a transformation $F : X \to Y$, then the induced operator F^* is adjoint, $F^* :$ Fun$(Y) \to$ Fun(X). This can be seen from the standard definition, the pullback: $(F^* f)x = f(Fx)$. This is important for the lifting of operators to the dual base.

One standard class of operators is composed out of *outer products* of vectors: $|\phi\rangle, |\psi\rangle \to |\phi\rangle\langle\psi|$, defined as $|\phi\rangle\langle\psi||\chi\rangle = \langle\psi|\chi\rangle \cdot |\phi\rangle$. In Haskell we get

```
outer :: (Vspace v, Hbase a) =>
  v -> HV (HV a) -> HV (HV a) -> v
(outer phi psi) chi = coax psi chi *> phi
--                  = chi (dual psi) *> phi
```

One simple and useful member of this family is a primitive projector $\hat{\mathbf{P}}_\alpha = |\alpha\rangle\langle\alpha|$, where α is an index. It may act on any vector, and it is defined by $\hat{\mathbf{P}}_\alpha|\psi\rangle = \langle\alpha|\psi\rangle \cdot |\alpha\rangle$. We must decide whether we need it to act on axes, kets, or on general bras. There are thus three differently typed instances of this operator.

```
axproj::(Hbase a) => a -> HM a
axproj alpha ax = ax alpha *> axis alpha
ktproj::(Hbase a) => a -> HM (HV a)
ktproj alpha kt =
       kt (axis alpha) *> ket alpha
brproj::(Hbase a) => a->HM (HV (HV a))
brproj alpha br =
       br (ket alpha) *> bra alpha
--   = br f *> coax f where f=ket alpha
```

(In a more consequent framework this should be one overloaded object; the class representing vector spaces should be appropriately augmented; this work is in progress).

In the section (2.2) we have shown how to construct co-vectors out of vectors, by the duality operations. The contravariance implies that having operators acting on co-vectors, e.g. on axes, we can reconstruct operators acting on kets, the recipe is universal, but we construct the adjoints! From an operator defined on axes, we construct one acting on kets by **boost**, introduced above. As examples we shall construct the operator of energy (quantum level) $\hat{N}$ (called: **level**) of an oscillator in the **N** basis, and the annihilator operator $\hat{a}$ (called **ann**) which decrements the excitation level n. They are defined by their actions on a one-component ket in this basis:

$$\hat{N}|n\rangle = n|n\rangle\,, \qquad \hat{a}|n\rangle = \sqrt{n}|n-1\rangle \quad (\text{for } n \geq 0)\,. \tag{3}$$

Their decomposition gives infinite sums

$$\hat{N} = \sum_{n=0}^{\infty} n|n\rangle\langle n|\,, \qquad \hat{a} = \sum_{n=0}^{\infty} \sqrt{n}|n-1\rangle\langle n|\,. \tag{4}$$

Here is the coding of **level**, starting with an auxiliary linear function **ax_level** which acts on axes, and has the following semantics: **ax_level (axis (N n))** gives **n*>axis(N n)**. Its definition is unique

```
ax_level ax a@(N n) = fromInteger n * ax a
```

The lifting of it to kets is given by **level = boost ax_level**. Following the same reasoning we may define the annihilation (lowering) operator $\hat{a}$, and its adjoint (or its hermitian conjugate), the "creation" operator, which increments the level of the oscillator: $\hat{a}^+|n\rangle = \sqrt{n+1}|n+1\rangle$, named **cre**. These infinite matrices have close functional representations **ann = boost ax_ann**, **cre = boost ax_cre**, where **ax_ann**, **ax_cre** are operators acting on axes. We take into account the contravariance of the lifting functor, which means that in the axes' space we must effectively define the adjoints:

```
ax_ann ax (N n) = isqrt n * ax (N (n-1))
ax_cre ax (N n) = isqrt(n+1) * ax (N (n+1))
-- (isqrt = sqrt . fromInteger)
```

It is easy to verify that **ax_ann** acting on a primitive **z=axis (N k)** produces a co-vector proportional to **axis (N (k+1))**, since non-vanishing **z (N (n-1)** implies a non-zero **(ax_ann z) (N n)**. In the ket space the duality reverses this behaviour.

We have the quantum oscillator in the computer in the form as abstract as possible, and we can solve several exercises from a quantum mechanics textbook by programming, for example show that **cre . ann** yields an operator equivalent to **level**, or that the commutator **ann . cre <-> cre . ann** is the identity. If we invest the knowledge that for the oscillator the operator of the spatial position x can be expressed as $x = (\hat{a}+\hat{a}^+)/\sqrt{2}$, we can easily derive the oscillator wave-functions (Hermite functions) from the recurrence relations involving $|n\rangle$, see our introductory article [23], where we have also proposed a very compact program which gives in a few lines the complete lazy perturbational solution of the anharmonic oscillator problem. The code is available from the author. The approach taken typically by physicists in this context, is a heavy use of computer algebra packages, not used for insight, but for generating numerical programs. Here our manipulations are formal, but we process computational structures rather than symbols. In this sense this paper is the continuation of the philosophy exposed in [24].

All operators in the let space can be lifted to their adjoints acting on bras, but the presented construction does not permit to derive the adjoint which would act on vector of the same species, by the standard relation: $\langle\alpha|T^{+}|\beta\rangle = (\langle\beta|T|\alpha\rangle)^{*}$. The *construction* of an adjoint is a non-universal procedure. Only in a concrete basis it reduces to simple operations, such as matrix transposing and complex conjugation, otherwise some other specific properties of the operator must be known, e.g., the fact that $\left(\frac{d}{dx}\right)^{+} = -\frac{d}{dx}$ on the domain of functions which behave sufficiently decently (vanish sufficiently fast) at the boundaries of the region which determines their scalar product. Here the algebraic data types are easier to manipulate than functional objects.

2.4 Some qubit operators

We return to quantum bits and their sequences, since *today* they are more important for computer scientists than oscillators, point particles, etc. The operators which transform kets: $|\psi\rangle \rightarrow |\psi\rangle' = \hat{A}|\psi\rangle$ must be linear and unitary (preserving the norm). In the classical concrete representation, where the state is a "concrete" vector $\binom{\alpha}{\beta}$, an operator is a 2×2 matrix. We shall use mainly projectors, and we will see that we might define composition of operators backwards, using their adjoints.

The unary "not" (Boolean negation) operator lifted to the domain of vectors (kets) should satisfy: $\mathbf{qnot}|0\rangle = |1\rangle; \quad \mathbf{qnot}|1\rangle = |0\rangle$. Its matrix representation is thus the Pauli σ_x matrix: $\mathbf{qnot} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. It is self-adjoint, and moreover it is an involution (its own inverse). Its more abstract representation is the "switching" operator $|0\rangle\langle 1| + |1\rangle\langle 0|$, and this is our implementation, which is a slightly modified set of functionals already presented in the section (2.3), but restricted to represent dyadic products of kets, elementary or not. Thus for any kets $|p\rangle$ and $|q\rangle$ we define $|p\rangle\langle q|$:

```
dyade :: (Vspace (HV a), Hbase a) =>
   (HV t) -> HV (HV a) -> t -> HV a
dyade p q = \ax -> p ax *> dual q
```

and for elementary $|\alpha\rangle$, where α is a state label, we have **`warp alpha beta = dyade (ket alpha) (ket beta)`**, which can be optimised into

```
warp alpha beta =
 \ax -> ax alpha *> axis beta
```

The projector $|\alpha\rangle\langle\alpha|$ is just **`warp alpha alpha`**. We named "warping" the dyade $|\alpha\rangle\langle\beta|$, since it "bends" one direction in the Hilbert space onto another one, but this is not a standard term. The quantum negation is

```
qnot :: HM Qubit
qnot = warp B0 B1 <+> warp B1 B0
```

The π-phase shifter σ_z (another Pauli operator) represented by the matrix $\left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$ takes the form

```
sigz :: HM Qubit
sigz = proj B0 <-> proj B1
```

and the sum

```
ax_had = sqrt 0.5 *>(qnot <+> sigz)
```

produces the matrix $H = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$, the Hadamard operator, which performs the transformations $|0\rangle \rightarrow (|0\rangle + |1\rangle)/\sqrt{2}$, and $|1\rangle \rightarrow (|0\rangle - |1\rangle)/\sqrt{2}$, used further to build entangled pairs, to construct the quantum Fourier transform, etc. An arbitrary (real) rotation which transforms, say, $\binom{1}{0}$ into $\binom{\cos(\theta)}{\sin(\theta)}$: $\left(\begin{smallmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{smallmatrix}\right)$ has the representation

```
rot theta = cos theta *> id <+>
  sin theta *> (warp B1 B0 <-> warp B0 B1)
```

where the second term is proportional to the third Pauli matrix, $\sigma_y = \left(\begin{smallmatrix} 0 & -i \\ i & 0 \end{smallmatrix}\right)$, most papers on quantum computing traditionally omitting the imaginary factor.

Note the — already observed — contravariance of the operator construction, in view of the fact that operators are functions which do something to arguments of their arguments. The operators defined above, **`qnot`**, **`ax_had`** etc. act on axes. We have to **`boost`** them so that can work on kets. Suppose that we shall sequentially act on a ket $|\psi\rangle$ with two operators, say, first with A (**`opa`**), and then with B (**`opb`**). The computation: $|\chi\rangle = B\,A|\psi\rangle$, which can be graphically depicted as shown on Fig. 2, is implemented as follows. First we define the operators **`ax_opa, ax_opb`** acting on co-vectors (axes), and at the end we **`boost`** them:

```
opa = boost ax_opa
opb = boost ax_opb
```

This "quirk" will be very important for the construction on operators acting on tensor products, which are multi-linear. So, we have

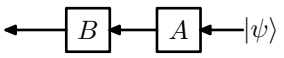

Figure 2: Chain of operators

```
chi = boost opbx (boost opax psi) =
      psi . opax . opbx =
      boost (opax . opbx) psi
```

That's why on the Fig. 2 the operators acting on kets are applied as drawn — from right to left, which is the opposite convention to one found in many papers on quantum gates, etc. But *this* drawing convention corresponds better to the standard (Dirac) notation, and we shall keep it for mnemonic purposes.

3. CONSTRUCTION OF COMPOSITE SYSTEMS

3.1 From Cartesian to tensor products

The construction of a classical system with many degrees of freedom, such as two rotators, or an oscillating particle with spin, is based on the simple set product: the system state is described, say, by a two-valued variable ***and*** with its excitation level. In general, we can — in principle — build a compound **`Hbase`** using the Cartesian product constructor:

```
data Qbase = Q Qubit | ...
           | CP Qbase Qbase
```

A common truth in quantum physics is: the joint quantum state of two independent systems is their tensor product. For a modern discussion of this issue see [25], but the book [22] and many others provide a complete discussion of the related mathematics. See also the rich Web site of John Baez [26]. The forms above, involving **`CP`** will not be used at all.

If we want to construct *elementary* two- (or more, but practically restricted to few) sub-system states, say, $|0\rangle|1\rangle$, we may start with multilinear primitives, e.g.,

```
ket_2 alpha beta = \ax1 ax2 ->
   (ax1 alpha)*(ax2 beta)
```

In general, if a ket is a linear function defined on axes, a tensor product of two (or more) kets is a bi-linear (multi-linear) function of two or more axes: if **`kt1 = \ax->ktf1; kt2 = \ax->ktf2`**, then **`kt1<*>kt2 = \ax1 ax2->ktf1*ktf2`**, and this should be appropriately generalized to multi-linear forms. Knowing that our functions will need many arguments, it is good to define more general Vector Space instances, e.g.:

```
instance (Vspace b) => Vspace (a->b)
 where
  vZero v = vZero
  f <+> g = \x -> f x <+> g x
  f <-> g = \x -> f x <-> g x
  (a *> f) x = a *> (f x)
```

where the lifted arithmetic operations are defined recursively. The tensors are defined with the aid of the outer multiplication operator **`(<*>)`**, and they use seriously the multi-parametric classes with functional dependencies [27] in order to be sufficiently universal, but concrete enough so that the user doesn't need to put concrete type signatures everywhere. We define

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

where the functional dependency means that the type of $(p+q)$-linear tensors can be deduced from the p- and q-linearity of the factors. Scalars are natural tensors:

```
instance (Vspace v)=>Tensor Scalar v v
 where
  s <*> v = s *> v
```

and the most important recursive type constraint is

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

so, now we can construct **`ket2 = ket B0 <*> ket B1`**, and use it in our calculations. It is easy to prove that the tensor product is associative, although non-commutative. Instead of $|\psi\rangle \otimes |\phi\rangle$ we may write $|\psi\rangle|\phi\rangle$, or $|\psi\phi\rangle$.

The tensor product of states is an "irreversible operation" in the sense that in general it is not possible to extract one subsystem, although by performing a partial measurement (applying the vector to an incomplete set of Hbase arguments), the arity of the state function is reduced. The result is (usually) not normalized, and needs thus some re-interpretation, very important from the measure point of view. If a given bi- or multi-system state is not a single tensor product but a sum thereof, for example if $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle|0\rangle - |1\rangle|1\rangle)$, then this extraction of a single subsystem *is not possible at all* without destroying the quantum structure of the state. We say that the two subsystems are **entangled**. They form a whole, even if the two subsystems are separated in space by a large distance. This conceptual problem is of utmost importance, and it is discussed in almost all general papers introducing quantum computers. For a thorough discussion see [28] [5]. We cannot pursue this topic here, we signal only that because of the entanglement ***a complete simulator of a quantum system cannot be modularised into small, local units, each dealing with a small local sector of the global state***. The full state is shared. This non-separability is true for *any* model of a quantum system. Does the functional programming have any advantages wrt. modelling approaches which use bit strings and complex arrays? We are tempted to say: yes. The construction of tensor products in function spaces is more *natural* than for classical data structures. The implementation of entangled states is as simple as possible, while such a construction which uses pairs of data items representing single qubits, see e.g. [12], might be considered (from the physicist point of view) somehow artificial, although easier to manipulate.

[5] We thank one of the Reviewers for this reference.

3.2 Dual tensors

This section is rather short, but the issue is involved, and it requires more work. In order to compute scalar products we need dual bases also for composite systems. Passing from such kets, or from any combinations thereof to axes of known, low arity is relatively simple. We have to conjugate the result of the transposition

```
(transp_2 ktp) alpha beta =
  ktp (axis alpha) (axis beta)
```

where **`ktp`** is a 2-product ket, and **`alpha`** and **`beta`** are the appropriate Hbase elements. We see that a compound axis is also a bilinear function, and doesn't involve any "classical" Cartesian product of the associated Hbase labels. In general, the definition above may be simplified to a combinator form: **`transp_2 = (transp .) . transp`**, and a general transposed to a multi-ket

```
transp_n ktp q1 ... qn = \q1 ... qn ->
 ktp (axis q1) ... (axis qn)
```

may be represented as

```
transp_n = (transp_n1 .) . transp
```

where **`n1`**$\equiv$**`n-1`**. We have **`dual_2 = (conj .) . transp_2`** and in general the conjugation of **`transp_n`** involves n compositions, which is not nice.

For the scalar products (and the squared norms) we may begin with those forms acting on axes, but the procedure is effective on finite bases only, since it involves the summing of the complete set of projections. For qubits:

```
axprod2 ax2 bx2 =
 sum [ax2 x y*conjugate (bx x y)|x<-ql,y<-ql]
-- where
ql=[B0,B1]
```

with obvious generalization for higher rank axes. The norm of a 2-ket for the **`Qubit`** system is given by

```
norm2_2 ktp = axprod2 ax2 ax2 where
 ax2 = dual_2 ktp
```

Thus, we can compute the probability amplitudes of compound 2-, and higher, with known rank vectors quite easily, but the technique becomes difficult, and infinite bases, although possible to deal with, require a special treatment. Still, our functional approach offers an advantage, the universality and simplicity of notation, but the issue is difficult in any model.

3.3 Operators on tensor product states

Mathematically the tensor product of $\hat{A}_1$ which acts on $|\psi_1\rangle$, and $\hat{A}_2$ concerned with the second subsystem, is the operator $\hat{A}_1 \otimes \hat{A}_2$ whose semantics is the following:

$$\big(\hat{A}_1 \otimes \hat{A}_2\big)\big(|\psi_1\rangle \otimes |\psi_2\rangle\big) = (\hat{A}_1|\psi_1\rangle) \otimes (\hat{A}_2|\psi_2\rangle)\,. \qquad (5)$$

The implementation seems quite complicated, especially if we think already that the vectors which will be processed directly by the functionals defined in the program are in fact co-vectors (axes); we will have to boost multi-linear functions.

In different words: we have a set of "input", and a set of "output lines", like on Fig. 3, and we have to construct *one* object which performs this transformation. It is interesting to observe that when

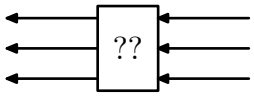

Figure 3: Operator on a composite state

we define such transformation acting on kets, (single kets which are tensorial, i.e., multilinear), the argument of the operator provides structurally a "continuation", and the composition of such operators is stylistically a little similar to the CPS programming.

Constructing the product of two operators is almost straightforward:

```
boost_2 ao1 ao2 ktp  =
  \ax1 ax2 -> ktp (ao1 ax1) (ao2 ax2)
```

where **ao1** and **ao2** are operators acting on single axes, and **ktp** is a 2-ket. Attention, the type checker will accept **ao1<*>ao2**, but the result is wrong. Although operators form a vector space, its tensorial structure is different from functionals interpreted as vectors. (Our notation and the type classes are being currently revised). Such factorised object can be depicted as on Fig. 4.

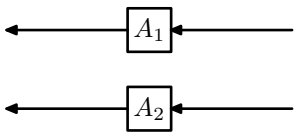

Figure 4: Tensor product of operators

The recursive construction of N-argument operators from lower-rank objects is not completely trivial. The definition of **boost_2** above may be reduced to

```
boost_2 ao1 ao2 = (boost ao2 .) . boost ao1
```

and for a product of rank 1 tensors, **boost_n**, defined as

```
\ktp -> \ax1 ax2 ... axn ->
 ktp (ao1 ax1) (ao2 ax2) ... (aon axn)
```

reduces to **(boost_n1 ao2 ... aon .).(boost ao1)**, where **n1**≡**n-1**, similarly to the reduction of duals to composites.

The multiplication of arbitrary composite tensors is clumsy, simple to do when tensors are *explicitly* given, and their rank is small. We have reached the zone where the structure of standard Haskell is not as convenient for us as before. In a realistic example, dealing with many qubits, we will have to construct functions with a huge number of arguments, and this is more difficult to digest than, say, manipulating an array with thousands of elements. There are no fundamental obstacles to that, but from the practical perspective the situation deserves some thoughts concerning the modularization and the compilation of those programs.

We need to compose functions with various arities, and — as show the circuit examples below – we need some generic techniques permitting to rearrange arguments of such objects. As the construction of multi-linear objects, duals, etc. is ***regular***, with recursively defined types and arities, the construction of appropriate combinators is not too difficult, but the results are ugly. We are convinced that several problems can be solved with the template extensions to Haskell[29]. Our work on the application of template tools has just begun.

4. MEASUREMENTS

In a quantum system any attempt to find out the information hidden in an unknown state will modify this state. Thus, this measuring process must be included in the theoretical model of a quantum system and of the information flow therein, if it is to be complete enough so as to deserve the name of 'simulation'. If one begins with concrete bit matrices which may be regarded, copied and transformed at will, the model is already "too classical" in the sense that we can "cheat" by looking inside it.

Bird and Mu propose that the quantum registers undergo a monadic, structured sequence of operations, giving to programs written in such a style an imperative feeling. We have just transformations of functions. The full state is always explicit, but the information within is well hidden.

4.1 Final computed results

As long as we stay within the quantum framework, *all* measurements (generation of numerical results) reduce themselves to computing of the mean values of some self-adjoint operators $\hat{A}$ in a state $|\psi\rangle$, which is denoted by $\langle\psi|\hat{A}|\psi\rangle$. One reads that the quantum measurements give us the probabilities of the components $|\alpha\rangle$ found in a given state, but this — according to (2) — is also an average of a self-adjoint operator, of a projector:

$$|\langle\alpha|\psi\rangle|^2 = \langle\psi|\big(|\alpha\rangle\langle\alpha|\big)|\psi\rangle\,. \tag{6}$$

Finally, the model gives us the probabilities, they are numbers, and we may stop. If we decide to go further, the remaining work is a "normal", classical (albeit non-deterministic) computation: we use some random number generator in order to generate the instances of the concrete classical configurations, according to the prescribed probabilities, see [11] for a functional framework for such a procedure. Here Haskell does not offer any particular advantage over other languages, apart from the elegance of notation.

In order to get some results methodologically meaningful, we must repeat the simulated experience many times. Unless we are sure that the result of a quantum process is either a value "↑" or "↓", and not an arbitrary superposition thereof, *one* individual experience conveys almost no information. This means that we must operate from the beginning on ensembles of many identically prepared quantum systems, and to use a random number generator many times, in order to gather a meaningful statistics. On the other hand, many serious algorithms in quantum computing are designed to generate "committed" (or almost) states corresponding to classical configurations, and not to their superposition. In such a way *one* measurement should provide a definite answer.

Our package uses random number generators to produce *arbitrary* (finite, tabularized) discrete distributions, and it may work even in some cases of infinitely dimensional bases, provided that the probability amplitudes vanish sufficiently fast.

4.2 Entanglement example; mixed states

Let's analyze a classical Einstein-Rosen-Podolski problem [30]. Suppose that we have prepared two qubits in an entangled state $|\chi\rangle$

```
chi = sqrt 0.5 *>
  (ket B0<*>ket B0 <+> ket B1<*>ket B1)
```

We adopt the convention that in any context, in the definition above, or in the expressions $|ab\rangle$, or $\langle ab|$, always the *left* symbol (here: a) belongs to the "first" subsystem.

Now we say that the first qubit is sent to Alice and the other to Bob[6]. Alice measures her system, and obtains, say **B0**, which immediately "collapses" the configuration of the qubit owned by Bob to **B0** as well. But Alice is a functional programmer, and wants to simulate the procedure. She knows that **chi** is *somehow* shared by herself and Bob, that they belong to the same non-separable sector of quantum world, despite the current physical separation of the subsystems. What happens to **chi** when she measures her qubit? If we are to *program it*, we must be disciplined. For example, we are not allowed to say that "the state of the qubit owned by Bob collapses", we cannot code it.

The only possible procedure for Alice is:

- She constructs a projector for her qubit, say $|0\rangle\langle 0|_1$, where the subscript is a visual remainder.
- She cannot touch the second qubit. She makes the tensor product of her projector by the identity $\mathbf{1}_2$, and applies it.
- The filtered state is $|\zeta\rangle = \frac{1}{\sqrt{2}}|00\rangle$. The normalization factor gives the amplitude of a successful reduction. Of course, the experience may randomly produce $\frac{1}{\sqrt{2}}|11\rangle$. The factor $1/\sqrt{2}$ gives the probability amplitude for *this* reduction. The new state (duly normalized to 1; the probability amplitude generated at Alice's site has no meaning for him) is the vector Bob can use.
- The translation to Haskell of all this is a pure syntax, the operator which acts on **chi** (for the **B0** reduction) is **op = boost_2 (proj B0) id**, and this is all. A complete simulation must unfortunately compute all possible filterings, and classically it might be a costly procedure.

We shall reiterate a similar manipulation in the next section when discussing the simulation of the teleportation.

And now comes the touchy point. We said that Bob is left with the state **ket B0<*>ket B0** (or the other one). But how can he know that Alice performed the measurement? He cannot. He will get a random result independently of whether he uses the original state vector, or the vector reduced, but in the latter case the "randomization" takes place at Alice's site! The program which simulates Bob *must* know it. There is *one state* which propagates along the chain of linear transformations until the final measurement which produces a known, final state. Bird and Mu suggest an analogy between such streamlining of operations and the monadic chain in a purely functional framework. We may intuitively think of it as a chain of transformations of physical attributes of the system, while in fact we are just composing functions (like in a State Monad) acting on "virtual attributes". The difference between the Bird and Mu formulation, and ours is that they chain the operator applications, acting ultimately on a data item representing a register.

The situation changes when we want to decouple Bob from Alice through decoherence, a phenomenon which *surely* will influence the behaviour of a quantum computer attached to a classical interface. We put his subsystem in a statistical framework, saying that Bob will perform his measurements assuming that all possible measurements at Alice's site have been completed. The quantum theory says that Bob should use a *mixed state*. It is easy to see that in all *final, physical* formulae involving a state $|\psi\rangle$ one really needs its projector form $|\psi\rangle\langle\psi|$. It becomes $\sum_b |\psi\rangle\langle\psi|$, where b corresponds to the parameters of the Alice's subsystem. In the functional formulation this is relatively straightforward. We obtain an operator, which in the case of simple entanglement is equal to $\frac{1}{2}\mathbf{1}$, and in general, for a simple qubit averaged over:

```
bstate = qsum
 [outer (psi ax)(psi ax)|ax<-map axis [B0,B1]]
```

where **qsum** is a fold of **<+>**. If the implementation uses classical data structures to represent states, the construction of mixed states becomes clumsy. The reader should not consider this section as a philosophical divagation irrelevant to the main topic of the paper: the functional implementation of quantum entities. Its aim is to show the strength of quantum constraints on *any* implementation. If two subsystems interacted once, they will never be independent, and the program must operate upon a global state, since the interaction ***may*** produce entanglement. Our Haskell implementation makes it clearly visible.

5. QUANTUM CIRCUITS

5.1 Some elementary gates

We have seen already some "gates" (operators) on single qubits, such as the negation. From the Pauli matrices we can construct the rotations, phase shifts, etc., but in order to be able to *compute*, it is necessary to have some multi-bit, or rather multi-qubit operators, and some generic mechanisms to compose them. We know already how to make tensor products, and we know that the operators form a vector space.

The basic, and very strong requirement imposed on those gates is their unitarity: $A^+ = A^{-1}$, which implies reversibility. This means that a classical gate, say NAND which combines two bits-arguments in one-bit result is an illegal operator, it does not correspond to a physical evolution of a quantum system.

Thus, one can read sometimes that a legal operator must have the same number of input and output lines. This is a slight trivialization of the problem, of course there are quantum processes which create or annihilate particles, everything depends on the internal structure of these "lines". Two spins $1/2$ may transmute together into one rotator with spin 1. But for computing purposes, even a 1-to-1 process, a 1-bit function $f(x)$ may be illegal if it is not reversible. It has been shown (see e.g., [6]) that by adding extra "ballast" lines with the extra data frozen, all functions may be converted to bijections. For example, in order to construct an equivalent of a XOR gate, we add one output line, which copies one input. The result, whose standard graphical form is depicted on Fig. (5) is called the "controlled-NOT" gate, corresponds to the transition:

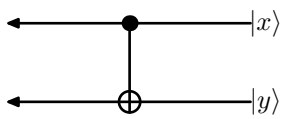

Figure 5: Controlled-not gate

$|x\rangle|y\rangle \rightarrow |x\rangle|x \oplus y\rangle$, and has the following definition:

```
cnot kt ax ay = p B0 + p B1 where
  p b = kt (qproj ax b) (xor (axis b) ay)
```

[6]These names became folkloric.

```
-- where
qproj ax b = ax b *> axis b
xor r = r B0 *> id <+> r B1 *> qnot
```

Notice that the ***simulated*** gate performs a measurement (filtering), since it splits the state explicitly into two projections, and it is unavoidable.

5.2 Example: Deutsch problem

One of the simplest algorithms specific to quantum processing is the solution of a toy problem proposed by Deutsch. Given an unknown one-bit function $f(x)$ find as fast as possible whether the function is constant, $f(0) = f(1)$, or not. Classically it requires two measurements. But if we manage to convert this function into a quantum operator, it may be applied to a particular superposition of states $|0\rangle$ and $|1\rangle$, and return some answer in one step. (Of course, this will need some filtering, but we have already accepted the fact that on genuine quantum systems it takes no time; the "two elementary applications" are executed in parallel. In a simulated model we won't obtain anything miraculous. We show this example just as an illustration how to compute with our abstract vectors. A more interesting example would be the Jozsa-Deutsch problem which concerns not one qubit, but a n-long quregister. This would require the usage of general tensor products, but other technicalities would be similar.

First, we will generalize the controlled-NOT gate to the operator

$$|x\rangle|y\rangle \to |x\rangle|f(x) \oplus y\rangle \,. \tag{7}$$

```
fcnot f k x y = p B0 + p B1 where
   p b = k (qproj x b) (xor (axis (f b)) y)
```

This is the central processing module within the circuit which solves the entire problem, and which is shown on Fig. (6). Two assigned

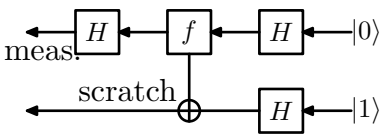

Figure 6: Deutsch problem

input lines: $|0\rangle$ and $|1\rangle$ are processed first by Hadamard transforms (the tensor products thereof, of course, as shown).

This part of the circuit takes the input into the combination

$$|0\rangle|1\rangle \;\to\; \frac{1}{2}\left(|0\rangle + |1\rangle\right)\left(|0\rangle - |1\rangle\right) \tag{8}$$

$$= \frac{1}{2}\left(|00\rangle - |01\rangle + |10\rangle - |11\rangle\right) \,. \tag{9}$$

The central module applies the function f. If it is constant, say $f(x) = 0$ for all x, the state changes into

$$\to \frac{1}{2}\left(|00\rangle - |01\rangle + |10\rangle - |11\rangle\right)$$

$$= \frac{1}{2}\left(|0\rangle + |1\rangle\right)\left(|0\rangle - |1\rangle\right) \,, \tag{10}$$

and if f is, say the identity, then we will obtain

$$\to \frac{1}{2}\left(|00\rangle - |01\rangle + |11\rangle - |10\rangle\right)$$

$$= \frac{1}{2}\left(|0\rangle - |1\rangle\right)\left(|0\rangle - |1\rangle\right) \,. \tag{11}$$

In both cases the *lower* line remains the same, but the upper, x line changes in a particular way. If we apply *to it* (the lower line is scratched) the Hadamard transform again, for f const the outcome is proportional to $|0\rangle$, and for the other case, to $|1\rangle$.

We will show the coding, but first a few words about the nonchalance of this derivation as seen from the programming perspective. What kind of mathematical object represents $f(x)$ in (7)? Is it a configuration label (Hbase), suggested by its presence in a ket, or a *number* 0 or 1? In typical presentations this problem is never explicit, the authors put or extract numbers into, or out of kets without any comments, they use "1" as a *numerical index*.

In our abstract framework we are constrained by the Haskell type system, and `f::Qubit -> Qubit`. Actually, the function f ***cannot*** be an operator on general quantum objects, it can do something only to a classical configuration, not to a superposition. We define two such objects, a mutating and a constant functions: `fmut = id`, and `fcst = const B0`. The circuit is represented by the following construction:

```
circuit f =
  let in1 =
     (boost2 ax_had ax_had) (ket B0 <*> ket B1)
  in  boost2 ax_had id (fcnot f in1)
xout = circuit id
yout = circuit (const B0)
```

It suffices to measure those last states: reduce e.g.

```
kmut = \ax -> xout ax arbitrary
kcst = \ax -> yout ax arbitrary
```

in order to find that if we freeze arbitrarily the second qubit (or if we average over it, which does not change anything), then the reduced states are proportional, `kmut` to $|1\rangle$ and `kcst` to $|0\rangle$. We have shown that the functional framework permits to code in Haskell *literally* the mathematical description of the problem. For readers less acquainted with quantum computing it serves also to show what kind of problems may be treated by quantum circuits; our objective is more pedagogical than technical.

5.3 Teleportation

This is another example of manipulation of compound, entangled states, showing how to transmit an unknown quantum state using two classical bits of information. The original state must be destroyed in this process, it is known from the *non-cloning theorem* [31] that a quantum state cannot be copied, i.e., there is no unitary operator which transforms, say, $|0\rangle|\phi\rangle$ into $|\phi\rangle|\phi\rangle$, where $|\phi\rangle$ is unknown. The interest of this example is mainly illustrative, although the relevant experiment has been performed with success. Even if the hardware would permit to extend the physical procedure to a macroscopic number of qubits, we would be very far from Star Trek, since first the transmitter and the receiver would have to be supplied with an adequate number of entangled units. Alice, the transmitter, disintegrates and measures the unknown state using her pool of entanglement resource, and sends the classical information to the receiver. Bob exploits these data to construct a machine which converts his entanglement pool into a copy of the transmitted object. Suppose that a shared source produced an entangled state $|\chi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ (called sometimes an axis of the Bell's basis), which can be done applying to $|00\rangle$ first *one* Hadamard operator, and then a `cnot` gate, as shown on the Fig. 7. Alice gets one qubit from this pair, Bob another one. Alice possesses an *unknown* qubit, in a state $|\phi\rangle = a|0\rangle + b|1\rangle$, so the state of the world is $|\psi\rangle = |\phi\rangle \otimes |\chi\rangle$. (Suppose that in a form $|\alpha\beta\gamma\rangle$ the first two labels belong to Alice). Alice applies to her sector the left part of

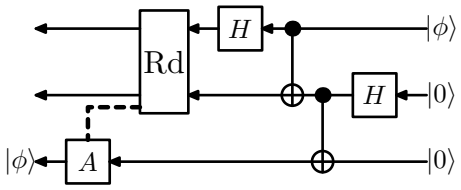

Figure 7: Teleporting circuit

the circuit shown on Fig. 7, where the dashed line represent the transmission of some classical information, which configures the operator A. (Of course nothing, i.e., the identity is applied to the Bob's part). It is easy to show that the state before the measurement ("Rd") becomes

$$\begin{aligned} |\omega\rangle &= \frac{1}{2}\big(|00\rangle(a|0\rangle + b|1\rangle) + |01\rangle(a|1\rangle + b|0\rangle) \\ &+ |10\rangle(a|0\rangle - b|1\rangle) + |11\rangle(a|1\rangle - b|0\rangle)\big)\,. \end{aligned} \quad (12)$$

The measurement at Alice's site produces one of four possible results, leaving the state in one of possible reduced forms $|00\rangle, \ldots, |11\rangle$, selecting thus $a|0\rangle + b|1\rangle$, to $a|1\rangle - b|0\rangle$. The simulator has to compute all the reductions, and then to choose randomly one, with the appropriate probability distribution. The choice 1 of four means 2 bits of information. They drive the choice of the operator which Bob will apply to his qubit (leaving the Alice sector alone):

1. The first case is the identity. Nothing to do, qubit may be processed further.
2. In order to convert $a|1\rangle + b|0\rangle$ to the wished state, Bob applies the **`qnot`** operator.
3. The third state $a|0\rangle - b|1\rangle$ needs **`sigz`**, and finally
4. The last one requires **`warp B1 B0 <-> warp B0 B1`**.

(The forms **`qnot`** etc. should be lifted from the domain of axes to kets.) The numerical answer confirms only that an effective procedure which mirrors the derivation on paper, can be coded in very few lines. We begin by introducing **`k0=ket B0, k1=ket B1`**, and choosing some values for **`a`** and **`b`**. Then we construct the entangled state through the first par of the circuit, and we multiply it tensorially by the "unknown" state:

```
phi = a*>k0 <+> b*>k1
psi = phi<*>(cnot.boost_2 ax_had id) (k0<*>k0)
```

Now we have to apply the upper left **`cnot`** gate, followed by the Hadamard operator. This is not obvious yet, since we know how to apply a binary gate to a binary ket, and here we have 3 lines. The Fig. 8 at the left would be an easier case. If **`op2`** is a bi-

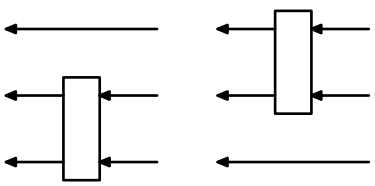

Figure 8: Two non-factorizable circuits

ket operator: **`op2 kt = \ax1 ax2 -> ...`**, its embedding into **`op3`** which takes 3 lines would be: **`op3 kt = op2 . kt`**. Our situation corresponds to the right diagram on Fig. 8, and we need to extract formally from a 3-ket its first two-sector:

```
\ay1 ay2 -> kt ay1 ay2 ax3
```

fixing the third line to **`ax3`**. This can be done with a combinator applied to a function of three arguments, which flips their order, putting the third one at the front: **`flip3 = flip . (flip .)`**. The destructor gate applied by Alice acts in a following way, producing the state **`omega`** given by the eq. (12):

```
cnot3 kt x1 x2 x3 = cnot (flip3 kt x3) x1 x2
omega = boost_3 ax_had id id (cnot3 psi)
```

The rest is just the measurement, and the reconstruction according to the variant instantiated by Nature (or by a random-number generator). The presence of fixed-arity combinators such as **`flip3`** is not nice, unfortunately the Haskell structure, its type system with the multiparametric classes makes it difficult to construct a general argument-permuting functionals for any arity, although it permitted to form general tensor products in an elegant way. Again, templates might be helpful here.

In this introductory paper we cannot show more elaborate examples. The constructions are not always readable, especially for computer scientists not familiar with the formal structure of quantum mechanics, but for others it might appeal by its "natural flavour", and they are fairly straightforward (although calling them an undergraduate exercice would be a mild exaggeration...). We underline that the idea is not to show some cute programming tricks, but to throw a bridge between the formalism of quantum mechanics and its representation within a functional program.

6. CONCLUDING REMARKS

It is difficult to say when we will have working quantum computers, but we are convinced that the paradigms of functional programming constitute a sound basis for their modelling, understanding, and also, in some possible context — their programming.

In this, preliminary work, we propose an abstract geometric framework permitting to define standard quantum entities as implementable functional objects. The level of abstraction is so high that we can offer a common style for the simulation of quite different quantum systems, and yet propose an effective coding, permitting to obtain some numerical results. Moreover, this genericity together with the strong typing discipline makes it more difficult to introduce errors in the program.

We believe, and we wanted to show that a modern, strongly typed and polymorphic functional language seems actually to be a fascinating tool for the implementation of quantum structures, although the Haskell type system seems a little too rigid, which makes it difficult to write functions acting on tensor products of arbitrary arity. As we mentioned, the template extensions might be helpful, permitting to work on *syntactic forms* as other people do on registers: pairs and lists.

Shall it be considered a practical tool for the simulation of quantum circuits? Probably not *yet*, our representation is more costly than the techniques based on arrays, the computations are more indirect. But it is "honest" in the sense that it is more difficult to violate the integrity of the simulated structures, to perform operations illegal from the quantum measurement perspective. This is good for the mental discipline of the programmer, and may provide a sound way of *representing/simulating* in the classical module of a future quantum computer its truly quantum parts.

On a conceptual note: in a more disciplined terminology *states are not measured*, they exist as a context for measuring the observables by providing a way to compute the averages $\langle\psi|\hat{A}|\psi\rangle$. "Measuring the state" means that the concerned observable is a projector on some basis. But **states and observables are not independent entities.** We may write the Schrödinger equation describing the state evolution in time: $|\psi\rangle \rightarrow \hat{U}(t)|\psi\rangle$, but this is conventional, in the so called "Heisenberg picture"[19] the states are never modified after their initial preparation; only the observables evolve with time. Since the only physically meaningful quantity is a bracket $\langle\psi|\hat{U}^+\hat{A}\hat{U}|\psi\rangle$, we may attribute the evolution to the operator: $\hat{A} \rightarrow \hat{U}^+\hat{A}\hat{U}$.

Thus, quantum states are **much** more abstract than their classical counterparts, and we hope that our representation underlines well this feature. The states include — through the possible measurements — several "virtual" possibilities, not always observed, like functions which may be, or not, applied to some arguments. States cannot be implicit, in a quantum reality it is very difficult to specify what a "side-effect" could mean... It is tempting to conjecture that the functional vision of quantum entities may be more closely related to the essence of Nature than the imperative models, with their explicit data structures, manipulated and modified at will.

7. ACKNOWLEDGEMENTS

We thank Jan Skibiński for interesting discussions during the preliminary work on this subject. Unfortunately his unpublished work is difficult to retrieve, see [32].

8. REFERENCES

[1] D. Deutsch, R. Jozsa, *Rapid solution of problems by quantum computer*, Proc. Roy. Society, **A400**, (1992), pp. 553–558.

[2] Peter Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, Proc. Symp. on Fundamentals of Computer Science, Los Alamitos, IEEE Press, (1994), pp. 124–134.

[3] L.K. Grover, *Quantum Mechanics Helps In Searching For a Needle in a Haystack*, Phys. Rev. Lett. **79**, (1997), p. 325. Also: L.K. Grover, *A fast quantum mechanical algorithm for database search*, Proc. 28th ACM Symp. on Theory of Computation, (1996), p. 212.

[4] D.S. Abrams, S. Lloyd, *Quantum algorithm providing and exponential speed increase in finding eigenvectors and eigenvalues*, Phys. Rev. Lett. **83**, (1999), pp. 5162–5165.

[5] Richard P. Feynman, *Simulating physics with computers*, Int. J. Theor. Phys. **21**, (1982), pp. 467–488.

[6] John Preskill, *Quantum Information and Computation*, Lecture Notes for Physics 229, California Institute of Technology, (1988).

[7] B.M. Boghosian, W. Taylor, *Simulating quantum mechanics on a quantum computer*, Online preprint quant-ph/9701019, (1997).

[8] D.G. Cory, et al., *Quantum Simulations on a Quantum Computer*, Phys. Rev. Lett. **82**, (1999), pp. 5381-5384.

[9] C. Zalka, C. *Efficient simulation of quantum systems by quantum computers*. Online preprint quant-ph/9603026, (1996).

[10] S. Lloyd, *Universal Quantum Simulators*, Science **273**, (1996), pp. 1073–1078.

[11] Shin-Cheng Mu, R. Bird, *Functional Quantum Programming*, 2nd Asian Workshop on Programming Languages and Systems, KAIST, Dajeaon, Korea, (2001).

[12] Amr Sabry, *Modeling Quantum Computing in Haskell*, These proceedings: Haskell Workshop, Uppsala, (2003).

[13] Bernhard Ömer, *Procedural Formalism for Quantum Computing*, (1998), available from **`http://tph.tuwien.ac.at/~oemer`**.

[14] Paolo Zuliani, *Quantum Programming*, PhD. thesis, St. Cross College, Univ. of Oxford, (2001).

[15] Julia Wallace, *Quantum Computer Simulation - A Review; ver. 2.0*, Univ. of Exeter tech. report, (1999), see also the site **`www.dcs.ex.ac.uk/~jwallace/simtable.html`**, (2002).

[16] X. Wang, A. Sørensen, K. Mølmer, *Multibit Gates for Quantum Computing*, Phys. Rev. Lett. **86**, pp. 3907–3910, (2001).

[17] E. Knill, *Conventions for Quantum Pseudocode*, LANL Rep. LAUR-96-2724.

[18] P.A.M. Dirac, *The Principles of Quantum Mechanics*, Clarendon press, Oxford, (1958).

[19] Albert Messiah, *Quantum Mechanics*, Dover Pubs., (2000), or any other, relatively modern book on quantum theory.

[20] G.J. Sussman, J. Wisdom, with M.E. Mayer, *Structure and Interpretation of Classical Mechanics*, M.I.T Press, (2002).

[21] T.C. Ralph, W.J. Munro, G.J. Milburn *Quantum Computation with Coherent States, Linear Interactions and Superposed Resources*, Univ. of Queensland e-print arXiv:quant-ph/0110115, (2001).

[22] Daniel Kastler, *Introduction à l'électrodynamique quantique*, Dunod, Paris, (1960).

[23] Jerzy Karczmarczuk, *Scientific Computation and Functional Programming*, Computing in Science and Engineering, Vol. 1, (2001), pp. 64-72.

[24] Jerzy Karczmarczuk, *Generating power of Lazy Semantics*, Theor. Comp. Science **187**, (1997), pp. 203–219.

[25] Diederik Aerts, Ingrid Daubechies, *Physical justification for using the tensor product to describe two quantum systems as one joint system*, Helvetica Physica Acta, **51**, (1978), pp. 661–675.

[26] John Baez, Web site **`math.ucr.edu/home/baez/photon/schmoton.htm`**.

[27] Mark P. Jones, *Type Classes with Functional Dependencies*, Proc. of the 9-th European Conf. on Programming, ESOP'2000, Springer LNCS **1782**, Berlin, (2000).

[28] John S. Bell, *Speakable and Unspeakable in Quanatum Mechanics*, Cambridge Univ. Press, (1989).

[29] Tim Sheard, Simon Peyton Jones, *Template meta-programming for Haskell*, ACM, Haskell Workshop, Pittsburgh (2002), pp. 1–16.

[30] A. Einstein, B. Podolski, N. Rosen, *Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?*, Phys. Rev. **47**, (1935), p. 777.

[31] W.K. Wootters, W.H. Zurek, *A single quantum cannot be cloned*, Nature **299**, (1982), p. 802.

[32] Jan Skibiński, *Haskell Simulator of Quantum Computer*, **`web.archive.org/web/ 20010630025035/ www.numeric-quest.com/ haskell/ QuantumComputer.html`**