Quantum Structures and Functional Programming

Jerzy Karczmarczuk

University of Caen, France

Abstract

We present a framework for specifying and solving computational problems in standard quantum mechanics using the purely functional, lazy language Haskell. We insist on a fairly high abstraction level; quantum states and operators are opaque functional objects, and their semantics is defined — as far as possible — independently of their concrete representations (the chosen base in the concerned Hilbert space). We show how to construct effectively the states for composed systems, and we present a toy model of quantum circuit toolbox. We exploit laziness in order to show how some perturbational algorithms become incredibly compact, yet effective.

This is a preliminary, incomplete draft, just for friends, and for my own assessment of what works, what is wrong, where are my technical problems, and what do I really want!

Keywords: Haskell, Quantum Physics, Vector spaces, Dual bases, Operators, Quantum gates, Perturbations, Automatic Differentiation, Lazy Programming, Multi-parametric Classes.

1 Introduction to Quantization

1.1 Quantum world is "abstract", thus functional

Despite the huge amount of code written and used by physicists, quantum mechanics is notoriously difficult to *model* on a computer. By models we mean here programs which operate upon "objects", with "physical" attributes (position, speed, etc.), with equations of motion resulting from some local dynamics of the system, sometimes with explicit probability distributions, but in other cases with statistical properties resulting from the chaotic dynamics, etc. This "realistic" simulation approach is rarely exploited within the domain of quantum physics. Yes, we can solve the Schrödinger equation, perform symbolically some awful perturbation expansions, visualize some interference patterns or diagonalize some matrices. We can even do it at home, using some popular numeric and visualisation software [1]. Still, we don't know really how to '*put a quantum system into a computer*' in a methodologically sound and intuitive framework, although all the relevant mathematical principles have stabilized many years ago, see [2, 3], and several other books with similar titles.

With the advent of "quantum computing" theory, see e.g., [4], and the elaboration of algorithms based on information transmission by quantum structures: quantum cryptography, teleporting, factorization of huge integers, etc., the recognition of this difficulty reached the realm of computer science, and favorized a somehow formalistic approach to the description of quantum systems. Researchers underline the massive parallelism of quantum algorithms, the exponential dependence of the number of states on the size of the system, the decoherence problems, etc. very hard technicalities.

But some deep, conceptual problems remain. The theory gives us statistical predictions only, but a simulation approach is based on the implementation of individual, unitary systems. The intrinsic non-determinism (and later the non-locality, and other epistemologic monstruosities…) makes the whole domain difficult to model in a naïve way.

Moreover, our representation of the quantum world is inherently *abstract*. Numeric quantum codes are used to diagonalize matrices, or to solve the Schrödinger equation, but neither concrete operators: matrices or differential ops, nor wave functions in space represent physical objects; they are just concrete representations of quantum "observables", and of the *state*, which is **not** directly measurable, but which is necessary in order to measure the physical properties of the system. In this sense, all "classical" simulations of quantum discrete systems (see e.g. the review [5]), i.e. collections of qubits, using vectors of bits and of complex numbers, are methodologically a little flawed. We think that a more abstract, yet practical formalism would be useful, mainly for our understanding, but also for *practical* purposes.

We shall attempt to code those abstractions using a functional language Haskell. We refuse to touch any philosophical issues in this text, the semantics of the word "abstraction" is comparable to many other similar categories of meaning in computer science: a concrete vector is a collection of components, a "classical" data structure. An abstract vector is an entity which is independent of the coordinate system which would specify those components, it is a geometric entity, not a concrete data structure. An abstract operator acts on, and transforms abstract vectors. They will be functional *objects*, but we shall not pursue the object-oriented methodology in this paper ; the relation between the OO programming and mathematical abstractions is still a little problematic. At any rate, an abstract vector will be a functional object which is concretized by its action on a coordinate frame. So, obviously, the entities we play with, are also some concrete representations, but, as the reader will see soon, their generality is incomparable with the usage of standard matrices.

1.2 Aims and contents of this work

This is not a "Quantum Computing" paper, although some examples of quantum information processing are provided. We present a small set of programming tools, coded in Haskell, permitting to manipulate formal quantum structures at the level comparable to standard student textbooks on quantum theory, and, as we said above, we insist on keeping the genericity of the formalism as high as possible. A good deal of the transformations between introduced entities is based on the *universal properties* (in the categorical sense) of their mathematical contents. ***So, our main target is the teaching, and comprehension of formal quantum-mechanical calculi*** assisted *directly* by a computer.

In a sense we develop a "symbolic" computation package using a functional language, but we underline the fact that we do not processs any symbols, but objects specified through their mathematical, operational properties. Those properties are introduced, and coded in a minimalistic way. One of main ambitions of this work is a peculiar methodologic honesty: the programming framework does not encourage the user to "cheat": if something cannot be measured or copied, then it cannot. We shall not confuse abstract entities, such as the labels chosen for the basic vectors in a Hilbert space, and integer numbers: 0, 1 etc., which can be manipulated arithmetically. Readers acquainted with the functional programming shall see some parallels between those quantum constraints, and the fact that one cannot copy or analyze the structure of a functional object. These analogies don't seem to be accidental.

We don't want to restrict our attention to notorious qubits. In our opinion this is an annoying fault of many introductory texts devoted to quantum computing: computer science-oriented readers who see just one, concrete model of one physical system have often severe problems with the understanding of common, underlying laws, and with the recognition of many isomorphisms and particularities which are natural for people formed as physicists. A qubit in an ion trap is very different from a qubit implemented as a polarized photon. Thus, we shall begin with a whole big class of potential physical systems. If in some future we will have to model real, physical quantum circuits, we shall not escape from the necessity of analyzing their interaction with the environment, and the qubits will not provide a complete description of the system anyway. Moreover, although the career of the Quantum Computing domain among computer scientists is due to such "pure algorithms" as the Shor factorization algorithm [6], see also [7, 8], we believe that technically *much* more important will be the work on simulation of general, different quantum system on quantum computers, as noted already by Feynman[9, 10, 11, 12], since this is badly needed by all modern technology. The same attitude has been expressed by Preskill [16]. This means that we shall need some sound framework for a *universal* quantum computer, universal not only in the formal sense, like the Turing machine, but able to model more or less seriously other quantum systems[13, 14], which physically are far from qubit processors.

We shall often exploit as a standard example the quantum harmonic oscillator, whose set of states is *infinite*, labelled by integers from 0 to ∞, because it provides a good test for the effectiveness (*not* efficiency…) of the proposed algorithms.

This presentation is by necessity rather simple and incomplete, just an introduction to a future work. The paper is addressed to non-physicists, although it requires some minimum of knowledge of quantum mechanics, at least some acquaintance with the "quantum computing folklore", some mathematical (algebraic) generalities, and a strong interest in this field. All information essential for our coding will be introduced incrementally in the text, but we cannot explain all the physics behind. We do not address the problems of efficiency, although we recognize the importance of this issue. A reasonably good acquaintance with the functional programming and with the language Haskell is assumed.

1.3 Manufacturing quantum systems

A classical system *is* [1] a set of observable states. A flip-flop (a one-bit, two-level system), which later will become a Qubit, has two states, say, **Up** and **Down**, or **B0** and **B1** (suggesting more Booleans than orientation…). A particle (a massive point) has a position **x** and a momentum. A 3D rotator has an angular position: two real numbers describing the (normalized) rotation axis, and its azimuthal angle. Sometimes systems which seem identical at a glance on these states, are not. 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 very different in both cases. The fact that for the oscillating point **x** and **p** are *bounded*, may change completely the mathematical description of two systems, e.g., a quantum oscillator has a discrete (Fock) basis, labelled by the number of elementary excitation quanta; a free particle doesn't.

[1] from the modelling perspective; we shall not start an ontologic divagation here…

Passing to the quantum description of such systems we have to take into acount the following:

- Each classical configuration, for example **Down**, or **x**, or (θ, ϕ)) should be consider as a *label*, a "vector index" of a vector in a metric (Hilbert) space. ***This is the most important assertion of the whole introduction.*** A quantum state is represented (*cum grano salis*; the norm is conventional, and the global phase factor is unphysical) by such a vector. A classical state might thus be considered as a *one component* of a possibly infinitely-dimensional quantum state, but this classical state has no vector space properties attached to it, so we prefer to treat is as an abstract label, a description of a chosen basis.

- Some classical description elements are superfluous in a nontrivial way. One cannot independently specify the position **and** the momentum, or the axis and the azimuthal speed of the rotator. They constitute *alternative representations*, or *different bases*. At the present stage we don't need to speculate about the Heisenberg relations. 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 representations are possible.

We can define thus some quantum 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
```

where we have invested the following knowledge:

- Each item defined at the right represents a vector "index" space. Alternative bases are variants of these data structures. Of course, 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.

[2] just to avoid a rather silly answer: ''ANY''. This is an important issue, but at this level we cannot discuss the properties related to the underlying spatial substrate (if it exists at all; there are plenty of bi-level quantum systems where the orientation does not play any significant role). In further examples we shall use labels **B0** and **B1**.

- Note that a zero-dimensional set (finite number) of index values implies a finite-dimensional vector space, and a one-dimensional set of, say, **X Double** specifies an infinite-dimensional (here even non-enumerable) vector space. Thus, the **Qubit** datatype having two instances: **Up** and **Down**, is the foundation of a two-dimensional space with the basis vectors $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$, or, using the Dirac notation: $|1\rangle$ and $|0\rangle$, or $|+\rangle$ and $|-\rangle$, or $|\uparrow\rangle$ and $|\downarrow\rangle$, etc. A basis state vector for a particle, denoted by $|x\rangle$ has a continuous set of components, for x in $\mathbb{R}^1$. The generalization to $\mathbb{R}^3$ is obvious.

- For a Rotator, the dual 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 (spin). A photon is a particle with $j = 1$, but it can represent a qubit as well: $m = \pm 1$, since for massless particles the value $m = 0$ is excluded.

- In the last example of the harmonic oscillator, we introduced another useful (Fock) basis, the number **k** in **(N k)** is the level of energy, or the excitation number (the number of quanta), see the section (2.5). This is not the full truth, the energy level k in $|k\rangle$ must be non-negative, while an **Integer** has no such restrictions, but we shall deal with such details in another way, see the next section, (2.1).

Depending on the user needs, those bases may be freely augmented. For the harmonic oscillator, we may introduce yet another alternative basis, say, **...| Ch Complex**, a complex number which represent a so called *coherent state* — an "almost classical" wave packet base in which any compatible quantum state can be developed as well. This is extremely important for physics, begins to interest computer scientists, but for us it will play a secondary role.

Those data structures have no specific mathematical properties ***yet***, they are just labels (related to, but not identified with the basic vectors tagged by them). We cannot add them, nor multiply them by numbers. In order to satisfy the superposition principle, we must define all the relevant math, making from our quantum states fully-fledged vectors in a metric space. In order to represent compound systems we must introduce some structuring mechanisms such as tensor products as well.

2 Basic Programming with Quantum States

2.1 Vector structure induced by the metric

As mentioned above, the concrete "classic" configurations, say **Up** or **(N 3)** are labels attached to vectors forming a basis for a given system. If the space is metric, we may ***postulate*** the existence of an appropriate sesquilinear[3] "scalar product" for these entities. Following Dirac symbolics we call this product the "bracket". For example, the form **bracket (N j) (N k)** reads in the standard notation $\langle j|k\rangle$, (this order is conventional!) and it 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.
```

[3]Sesquilinear: linear in one, and anti-linear in other argument, $o(\alpha x, \beta y) = \bar{\alpha}\beta o(x, y)$.

where **Scalar** is usually a complex number (**Complex Double**). In fact, the package is more parametric than described in this paper; we give some details in the Appendix (A), but the solution is not fully satisfactory because of the specificities of the Haskell type system. The default definition of the bracket specifies the orthogonality of the basis vectors:

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

This holds only for discrete bases, in the continuous case we would have the Dirac delta distribution, which is singular, and needs a more sophisticated formal apparatus in order to put it into the computer. Scalars are not numbers then, but operators, or distributions in the Schwartz sense; this is postponed to a future work, from now on we shall concentrate on discrete (not necessarily finite) systems. The default bracket is not the only one, and the instances may override it, 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 all the negative excitation levels. For the rotator state $|j, m\rangle$: $|m| \leq j$ must hold. There exist non-orthogonal bases as well (the coherent states for an oscillator is a good example thereof), and, as we know, **bracket (X x) (P p)** is a complex exponential $\exp(ipx)$ (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. Brackets will be use as fundamental bricks for the construction of quantum probability amplitudes.

Remarkably enough, although **Hbase** has no arithmetic properties, we can easily give them to *functions* over those "labels", by a known, standard construction, described in many books; our favourite is [15]. 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 codomain 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 Hvector b = b->Scalar

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

And now, defining a currified function over **Hbases**

```
axis :: (Hbase a) => a -> a -> Scalar

axis x = bracket x
```

we might assume that we got a representation of our quantum *basic* ***adjoint*** states $\langle\uparrow|$, $\langle n|$, etc. We can write

```
f = (2:+1)*>axis(N 3) <-> 4*>axis(N 1)
g = 0.5*>axis Up <+> (0:+2)*>axis Down
```

etc. [4], so, the "axes" are full-fledged vectors. But we will need more, they are not yet *general* members of a metric space. We don't know how to compute scalar products involving **`f`** or **`g`** from the combinations above. We will need some *linear* functions, the axes are auxiliary entities which cannot be linear because the **`Hbase`** has no associated algebra. However, we see here the power of a modern functional language, we have easily and effectively created an "abstraction". **`x = axis (N 4)`** is not decomposable, we cannot extract its only component, we can only check its value against another one, by acting with it on some **`(N k)`**, say, **`x (N 6)`**, 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 is yet far from obtaining a concrete experimental answer).

For a computer scientist the — perhaps — most important property of the mathematical structure imposed on the quantum states is that *physically* in the addition $|\chi\rangle = |\phi\rangle + |\psi\rangle$, or the filtering, say: $|\psi\rangle = \langle 0|\psi\rangle \cdot |0\rangle + \langle 1|\psi\rangle \cdot |1\rangle$ the two terms are evaluated in parallel, simultaneously, and the addition takes no physical time whatsoever! Obviously in our, or any other simulation of quantum processes, this is simply impossible, but this is the crucial point which distinguishes the complexity of the quantum processes from their classical shadows.

In the next step we define the dual base "ket"s: $|\uparrow\rangle$, $|n\rangle$, etc., as functions over our vector base (the axes). From now on, the term "vector" used generically will denote both axes and kets, but more specifically, we shall treat rather kets as vectors, and axes will be named *co-vectors*, in order not to forget the distinction between them.

The primitive kets, dual to elementary axes are:

```
ket alpha ax = conj ax alpha
```

where **`conj`** is the complex conjugation lifted to the functional domain: (**`(conj f) x = conjugate (f x)`**). 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. A linear combination of elementary kets (using **`<+>`**, **`<->`** and **`*>`**, since kets form also a **`Vspace`**) is an arbitrary vector $|\psi\rangle$. This is the principal face of the quantum states we shall work with.

We shall heavily use the combinatorial notation, simplifying **`f x = g x`** to **`f=g`**, and exploiting the standard combinators **`(.)`** and **`flip`**:

```
(f . g) x = f (g x)    -- Composer
flip f a b = f b a     -- "Flipper"
```

So, we can write:

```
axis = bracket
conj = (.) conjugate
ket = flip conj
```

There is one ingredient missing. How to compute the scalar product $\langle\phi|\psi\rangle$ of arbitrary $|\psi\rangle$ and $|\phi\rangle$, and in particular the squared norm $\||\psi\rangle\|^2 = \langle\psi|\psi\rangle$? We still don't have arbitrary "bra"s $\langle\ |$ with the same type as **`axis`**, and which can be considered as duals 'or adjoints, if you wish) of kets, so they can be constructed from them, and be their legal arguments, in order that **`kt (dual kt)`** give the correct answer 185. The construction is remarkably simple. The dual to a ket **`kt`** should be an axis, a function over **`Hbase`**. The following should hold

```
(dual kt) alpha = kt (axis alpha)  -- or:
dual kt = kt . axis
```

We may simplify the notation even more:

```
dual = boost axis                 -- where

boost = flip (.)
```

The functional **`boost`** will be used also in the section (2.3) and later. We can easily prove the validity of the construction: if **`kt = ket alpha`** is an elementary ket, then

```
dual kt beta = dual (ket alpha) beta
 = ket alpha (axis beta)
 = conjugate (axis beta alpha)
 = bracket alpha beta
```

which is correct. The linearity does the rest.

A conscious reader may observe that the construction seems unnecessarily complicated. For any **`axes`** objects $\langle\alpha|$ and $\langle\beta|$ we can compute $\langle\alpha|\beta\rangle$ as

$$\langle\alpha|\beta\rangle = \sum_\gamma \langle\alpha|\gamma\rangle\langle\gamma|\beta\rangle = \sum_\gamma \langle\alpha|\gamma\rangle\langle\beta|\gamma\rangle^* \,, \tag{1}$$

where γ is a **`Hbase`** index. In fact, for a finite base (e.g., the qubits), this is an effective procedure (and often the only one). 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 will 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 theoretical trick, never done by the Nature. Of course, 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.

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

$$P_\alpha(\psi) = |\langle\alpha|\psi\rangle|^2 \tag{2}$$

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 complete this section with the introduction of one missing object: the zero vector, not needed until now. If vectors are functions, then the definition of zero is simple: it produces the scalar zero by acting on anything. We complete the class **`Vspace v`** by: **`vZero ::  v`**, and define within its **`b->Scalar`** instance: **`vZero = const 0`**. It is not needed as an independent object if the field os scalars is equipped with zero, but sometimes it is useful for simplyfying some formulae.

[4] In Haskell the expression **`a:+b`** denotes a complex number $a + \mathrm{i}b$.

2.2 General bras and bi-dual base

It is becoming really boring, but for the simplification of some deriations we may need yet another space of co-vectors, this time identified with arbitrary bras $\langle\ |$, and implemented as functions over kets (the dual base). (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 = boost dual
```

Proof. A ket converted into a bra, acts on another ket producing a scalar.

```
coax kt kt1 = boost dual kt kt1
 = (kt . dual) kt1 = kt (dual kt1)
```

Exercise. Verify the sense of an alternative proposition

```
coax' = flip id . dual
```

where **`id x = x`** is the identity function. It yields **`coax' kt kt1 = kt1 (dual kt)`**. What is wrong with it?

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 alpha) kt =
        conjugate (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 conj`** itself transforms axes into bras, a simple exercise for the reader, who might prove also that the diagram on the Fig. 1 is commutative.

$$\begin{tikzcd}
|\mathrm{ket}\rangle \arrow[r, "\mathbf{dual}"] \arrow[dr, "\mathbf{ket}"'] & \langle\mathrm{axis}| \arrow[d, "\mathbf{coax}"'] \\
 & \langle\mathrm{bra}|
\end{tikzcd}$$

Figure 1: From kets to bras

On the other hand *there is no 'trivial' (universal) transformation* from arbitrary bras (including axes) to kets, the only way is the decomposition of a bra in a concrete basis, and the reconstruction of its dual from the coefficients. It involves thus a filtering, a part of measurement.

2.3 Operators

An operator is a function from vectors to vectors. Linear operators in quantum mechanics play primordial roles, some of them correspond to *observables*, other to symmetry transformers, and the whole evolution in the Schrödinger picture 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.

We may begin an explicit construction of some operators, departing from their action on the **`Hbase`** objets, e.g., saying that **`(N k)`** should become **`(N (k-1))`**, etc. 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 phi psi) chi = coax psi chi *> phi
--                  = psi (dual chi) *> phi
```

One simple and useful member of yhis family is a 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 $\hat{\mathbf{P}}_\alpha|\psi\rangle = \langle\alpha|\psi\rangle \cdot |\alpha\rangle$, so its implementation is easy, we must only decide whether we need it to act on axes, kets, or on general bras. There are thus three possible to define within our type system, differently typed instances of this operator, all immediate to construct.

```
axproj alpha ax = ax alpha *> axis alpha
```

Others:

```
ktproj alpha kt =
       kt (axis alpha) *> ket alpha
brproj alpha br =
       br (ket alpha) *> bra alpha
--   = br f *> coax f where f=ket alpha
```

The projectors (and other operators) form a vector space, the concerned class instance is

```
type Hmat b = Hvector b -> Hvector b
instance Vspace (Hmat 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 **`(.)`**.

In the previous section, (2.2) we have shown how to construct co-vectors out of vectors, by the duality operations. It is obvious then that having operators acting on co-vectors, e.g. on axes, we can reconstruct operators acting on kets. If we forget for a moment the fact that in fact we are defining the adjoints, the recipe is again trivial (universal). From an **`aop`** defined on axes, we construct an operator acting on kets by our old acquaintance **`boost`**.

How to represent the operator of energy (quantum level) $\hat{N}$ of an oscillator in the **N** basis, or the annihilator operator $\hat{a}$ which decrements the excitation level, and defined by their action on a one-component ket in this basis:

$$\hat{N}|n\rangle = n|n\rangle \, , \qquad (3)$$
$$\hat{a}|n\rangle = \sqrt{n}|n-1\rangle \qquad (\text{for } n \geq 0) \, , \qquad (4)$$

for arbitrary n? Their decomposition gives infinite sums

$$\hat{N} = \sum_{n=0}^{\infty} n|n\rangle\langle n| \, , \qquad (5)$$
$$\hat{a} = \sum_{n=0}^{\infty} \sqrt{n}|n-1\rangle\langle n| \, . \qquad (6)$$

We would have to operate with infinite matrices. Once again, the functional, parametric representation of states and operators takes into account the fact that effectively generated entities within the program have always a finite number of components, and the procedure becomes effective. But, attention, in a lazy language we can easily generate "infinite" objects, possessing an unlimited numer of components. The expression *effectively generated* means that only a finite part of such entities is instatiated. The user in principle can define such vectors as the coherent states for the oscillator, $|z\rangle = \sum_{n=0}^{\infty} \left(z^n/\sqrt{n!}\right)|n\rangle$:

```
coherent z = coh 0 1
 where
  coh n coeff = coeff *> ket (N n)
        <+> coh (n+1) (z*coeff/isqrt(n+1))
```

(where `isqrt = sqrt . fromInteger`), but the result is unusable. Handling of infinite series in a lazy language requires some conscious methods, see e.g. [17]. We shall use them later.

Here is the coding of the $\hat{N}$ operator. We begin with an auxiliary function **opn** which is linear, acts on axes, and has the semantics: `opn (axis (N k))` gives `k *> axis (N k)`. Its definition is

```
opn ax a@(N k) = fromInteger k * ax a
```

Finally, the lifting of the $\hat{N}$ (**level**) operator to kets is given simply by `level = boost opn`.

Following the same reasoning we may define the annihilation operator, and its adjoint (or its hermitian conjugate), the creation operator $\hat{a}^+|k\rangle = \sqrt{k+1}|k+1\rangle$. But beware, don't forget the contravariance of the lifting functors; we had no problems with $\hat{N}$, since this is a self-adjoint operator. If we define

```
oa ax (N k) = isqrt k * ax (N (k-1))
oc ax (N k) = isqrt(k+1) * ax (N (k+1))
```

then the *apparently lowering* operator **oa** acts on axes as a creator, it transforms `axis (N k)` into a vector proportional to `axis (N (k+1))`. The same paradoxal property is obeyed by **oc**, it is an annihilator in the axes' domain. But — as the tradition in quantum mechanics demands — we want operators acting on kets, and here the duality restores the order. In the section (2.4) we shall see more of that. Here we have

```
ann = boost oa
cre = boost oc
```

and this is all. All physics students in the world know that $\hat{N} = \hat{a}^+\hat{a}$. Proving that `cre . ann` yields an operator equivalent to **level** (and also that the commutator `ann . cre <-> cre . ann` is the identity) is more cumbersome, but it may follow the same formal, on-paper reasoning as the traditional one, found in quantum theory textbooks. We have the quantum oscillator in the computer in the form as abstract as possible, and we can solve several classical exercises from a quantum mechanics textbook by programming. We will not get any symbolic answers, though, unless we pass from the numeric domain of scalars and integers to some formal, symbolic algebra, which is beyond the aim of our current work.

Lifting of these operators to general bras is equally easy, we must boost them once more, but we obtain in such a way the adjoint of the operator. The presented construction does not permit to *derive* the adjoint which would act on vector of the same species, by the standard relation: $\langle\alpha|T^+|\beta\rangle = (\langle\beta|T|\alpha\rangle)^*$. This is quite obvious, the construction of an adjoint is a non-universal procedure, no categorical reasoning may help us here. Only in a *concrete*, discrete basis it reduces to simple operations, such as transposing and complex conjugation, otherwise some other *concrete* 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 at the boundaries of the region which determines their scalar product.

2.4 Some qubit operators

Material point dynamics, differentiation operators, infinite matrices corresponding to the annihilation operator, etc. are entities vital for physicists, but not so important *today* for computer scientists. We shall return to physics in the section (2.5), but the latter will demand the construction of models dealing with bits and their sequences.

Already at the one-qubit level, the transformation of state vectors is not entirely trivial, since the freedom of choice is limited. The corresponding operators, which transform kets: $|\psi\rangle \to |\psi\rangle' = \hat{A}|\psi\rangle$ must be linear and unitary (preserving the norm). In the classical concrete representation, where the state is a vector $\begin{pmatrix}\alpha \\ \beta\end{pmatrix}$, an operator is a 2×2 matrix. We shall use mainly projectors, and later we will see that in our functional style we define composition of operators backwards, using their adjoints, which is relatively easy to understand, but needs an additional explanation.

The unary "not" (Boolean negation) operator lifted to the domain of vectors (kets) should satisfy: $\text{not}|0\rangle = |1\rangle; \quad \text{not}|1\rangle = |0\rangle$. Its matrix representation is thus the Pauli σ_x matrix: $\text{not} = \begin{pmatrix}0 & 1 \\ 1 & 0\end{pmatrix}$. This is a self-adjoint operator, and moreover it is *involution* (it is its own inverse). In the domain of co-vectors (bras or axes)

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 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 optimized into

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

The projector $|\alpha\rangle\langle\alpha|$ is just `warp alpha alpha`. The dyade $|\alpha\rangle\langle\beta|$ with $\alpha \neq \beta$ is called *warp* since it "bends" one direction in the Hilbert space into another. The quantum negation is

```
qnot = warp B0 B1 <+> warp B1 B0
```

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

```
sigz = proj B0 <-> proj B1
```

and the sum

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

produces the matrix $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, the well known 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, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ into $\begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}$: $\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$ has of course 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, $\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$.

Note the — already noted — contravariance of the operator construction, in view of the fact that operators are functions which do something to their arguments. 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 acting on co-vectors (axes), and at the end we **`boost`** them:

```
opa = boost opax
opb = boost opbx
```

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

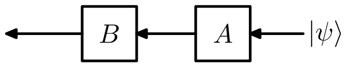

Figure 2: Chain of operators

```
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 most papers on quantum gates, etc. But *this* convention corresponds better to the standard (Dirac) notation, and we shall keep it.

2.5 Physical example

We shall solve now a conceptually simple, but non-trivial textbook problem: the perturbational corrections to the lowest energy state $|0\rangle$ of the anharmonic oscillator, with Hamiltonian: $\hat{H} = \hat{N} + \lambda\hat{H}'$ with $\hat{H}' = \lambda\hat{x}^4$, where λ is a small coupling constant, and $\hat{x}$ is the position operator. (We will adopt the unit values for the elasticity, mass, and the Planck constant; the energy has been shifted so that $E_0 = 0$, but we shall keep it for some time in order to have a general development algorithm.) For people interested in quantum computing this is a digression, but its aim is not the development of physics, but the demonstration of the power of lazy evaluation!

Perhaps it is worthwhile to say a few words about the *non*-perturbed system. The Hamiltonian of a classic oscillator is: $H_0 = \frac{1}{2}\left(p^2 + x^2\right)$ which, according to standard rules of quantization becomes the equivalent operator $\hat{H}_0 = \left(\hat{p}^2 + \hat{x}^2\right)/2$. Introducing $\hat{a} = \widehat{(\hat{x} + p)/\sqrt{2}}$ it is easy to show that $\hat{H}_0 = \hat{a}^+\hat{a} + 1/2$, and the energy levels are $E_0^{(n)} = \langle n|\hat{H}|n\rangle = n + 1/2$, where $n = 0, 1, 2, \ldots$. The spatial wave functions $\langle x|n\rangle$ which correspond to those energy levels are easy to compute on paper, and they can be computed almost directly by our package as well. We show, just for instruction, how to compute them numerically using the algorithm, which according to the current teaching standards is *par excellence* symbolic, and designed to be solved on paper. We *must* invest the knowledge about the standard representation of the momentum operator in the positional representation: $\hat{p} = \frac{1}{i}\frac{d}{dx}$. From: $\hat{a}|0\rangle = 0$, we deduce that $\psi_0(x) = \langle x|0\rangle$ obeys: $\frac{d}{dx}\psi_0(x) = -x\psi_0(x)$, so $\psi_0(x) = \exp\left(-x^2/2\right)$. But from the identity $\langle x|a^+|n\rangle = \langle x|\left(\hat{x} - i\hat{p}\right)/\sqrt{2}|n\rangle = \sqrt{n+1}\langle x|n+1\rangle$, we see that

$$\psi_{n+1}(x) = \frac{1}{\sqrt{2(n+1)}}\left(x\psi_n(x) - \frac{d}{dx}\psi_n(x)\right) . \qquad (7)$$

This is a differential, recurrent formula, which needs the derivative of ψ_n in order to compute ψ_{n+1}.

Of course we don't want to use any numerical approximations, nor involve any other tool other than our small Haskell library. We have included thus into it, our lazy automatic differentiation package [18], which permits to "lift" the normal numerical expressions within a program to a domain which structurally is an infinite sequence: the value of the expression together with its *all* derivatives, and which mathematically belongs to a simple, but non-trivial differential algebra. All typical arithmetic operations and elementary functions are overloaded for this domain. For all the necessary details see the appendix C. The procedure which constructs numerically the full set of Hermite functions $H_n(x)$ ***is literally*** the formula (7).

```
herm 0 x = exp(-x*x/2)
herm n x = (x*hh - df hh)/sqrt(2*dConst n)
 where hh = herm (n-1) x
```

It suffices to launch **`map (dVal . herm 30 . dVar)`**, where **`dVar`** constructs a generator (a "*differential variable*") of our differential algebra from a given numeric value, and **`dVal`** gets the "main value" of the resulting tower of derivatives, over a list of x values in order to generate the plot on Fig. 3. One of standard exercises in quantum mechanics is the comparison of the plot of $|H_n(x)|^2$ with the classical distributions in order to see what is the sense of the quasi-classical limit of quantum mechanics, so such exercises are methodologically useful, although rarely anybody needs the concrete numerical values of $H_{30}(x)$.

In finding the corrections we will not need the positional representation, only the $|n\rangle$ basis. It is utterly trivial to show that the first

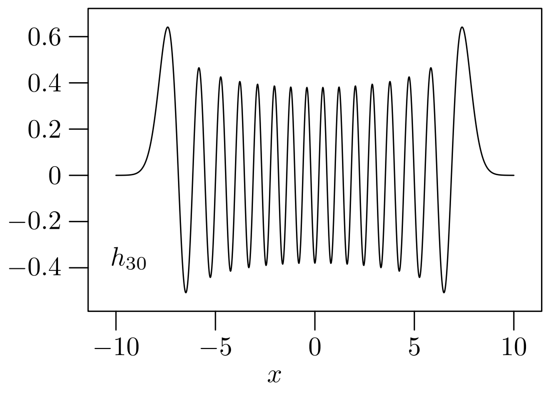

Figure 3: Hermite function H_{30}

correction to the unperturbed solution is equal to $\langle 0|\hat{H}'|0\rangle$, and the textbooks *sometimes* show a compact formula for the second term; usually this is left for the students, and it may occupy them for awhile… Computing the third-order correction is near the limit of the nervous resistance of a typical physicist; the problem is that for many interesting cases in technical applications of quantum physics such as molecular spectra, those, and much higher corrections *must* be computed.

The consequence of this is obviously a very intense exploitation of Computer Algebra packages, and the production of horrendous, multi-page formulae, unreadable (see e.g., [19]), and often badly optimized, just to convert them into Fortran programs, and compute numerically a few numbers. We have chosen the oscillator, and not, say, the Hydrogen atom, because of the simplicity, but the approach generalizes in a fairly transparent way (although the concrete formulae might become much more tedious…).

We show thus a complete solution of the stated above problem which gives the numeric solution directly. This will be probably the shortest program to do that which the reader could find anywhere, and it uses the series expansion described in [17], and presented in a few lines in the Appendix (B).

The Schrödinger equation which gives the energy E of a system in a state $|\Psi\rangle$ is: $\left(\hat{H}-E\right)|\Psi\rangle = 0$. We do not know E nor $|\Psi\rangle$. We may suppose, though, that if λ is small, the series expansion of $E = E_0+\lambda E_1+\lambda^2 E_2+\cdots$, and $|\Psi\rangle = |\Psi_0\rangle+\lambda|\Psi_1\rangle+\cdots$ makes some sense, even if the series is only asymptotic (divergent; this is the case here). Here $|\Psi_0\rangle = |0\rangle$, and we know that the unperturbed equation $\left(\hat{N}-E_0\right)|0\rangle$ is fulfilled. We reformulate the expansions above as folows:

$$|\Psi\rangle = |0\rangle + \lambda|\psi\rangle\,, \tag{8}$$
$$E = E_0 + \lambda E'\,, \tag{9}$$

where both $|\psi\rangle$ and E' are series in λ. The Schrödinger equation becomes

$$E'|0\rangle - \left(\hat{N}-E_0-\lambda E'\right)|\psi\rangle = \hat{H}'|0\rangle + \lambda\hat{H}'|\psi\rangle\,. \tag{10}$$

We are allowed to choose the normalization: $\langle 0|\Psi\rangle = 1$, which implies the orthogonality $\langle 0|\psi\rangle = 0$. This, after having written the scalar product of (10) with $\langle 0|$, and $\langle k|$ for $k \neq 0$, gives us:

$$E' = \langle 0|\hat{H}'|0\rangle + \lambda\langle 0|\hat{H}'|\psi\rangle\,, \tag{11}$$
$$\langle k|\psi\rangle = \frac{1}{-(k-E_0)+\lambda E'}\left(\langle k|\hat{H}'|0\rangle + \lambda\langle k|\hat{H}'|\psi\rangle\right) \tag{12}$$

The only final result needed is E'. The equations above are already effective algorithms able to compute two involved quantities, E' and $\langle k|\psi\rangle$, but a third one: $\langle k|\hat{H}'|\psi\rangle$ should better be eliminated. We achieve that through the substitution $\langle k|\hat{H}'|\psi\rangle = \sum_m H'_{km}\langle m|\psi\rangle$, where $H'_{km} = \langle k|\hat{H}'|m\rangle$, which transforms the equations above into

$$E' = H'_{00} + \lambda\sum_m H'_{0m}\langle m|\psi\rangle\,, \tag{13}$$
$$\langle k|\psi\rangle = \frac{1}{E_0-k+\lambda E'}\left(H'_{k0} + \lambda\sum_m H'_{km}\langle m|\psi\rangle\right) \tag{14}$$

It may seem rather useless to repeat here this classic derivation, but in no existing popular textbooks the reader will find such *algorithm*, since the laziness is here the crucial ingredient, and specialists in quanta rarely use functional languages. The rest is the coding, where we have simplified slightly the formulae, omitting the checks for k etc. negative; the matrix elements and the function **psi** vanish then.

As we have already mentioned, our code is more generic than presented above, the states etc. are parameterized by arbitrary scalars, which are equipped by a sufficiently rich arithmetic. Here the field of scalars is composed of expressions of the form **e0 :> e1 :> e2 :> ...** which represent the power series $e_0 + \lambda e_1 + e_2\lambda^2 + \cdots$. Such series are constructed lazily, and don't need any truncations. Here is the construction:

```
hp = x.x.x.x where
 x = (1/isqrt 2)*>(ann <+> cre)

elmat k m = hp (ket (N m)) (axis (N k))

ep = e0 :>
 sum [elmat 0 m * psi m | m<-[0,2,4]]
psi k = (elmat k 0 :>
 sum [elmat k m * psi m |
      m<-[k-4,k-2 .. k+4]])/
         (negate(fromInteger k):>ep)
```

The result is the series {0.75, -2.625, 20.8125, -241.2891, 3580.98047, -63982.8134766, …}. As seen, this time we used effectively the expansion of the quantum state in a known basis, but we *knew* from the structure of $\hat{x}$ that the number of non-vanishing matrix elements within the infinite sum is small.

Launching the program above results in a very bad, and very inspiring surprise. One gets 8, perhaps 9 terms, and the memory problems begin… The consumption of the memory ressources is exponential, because the lazily evaluated **psi k** is recursive, and each recursive call generates new instances of the **psi** thunk on the heap, thunks which become larger, and larger… One of the reasons why in scientific computations the lazy coding is rather unpopular, is that writing such codes needs a decent knowledge of specific optimization techniques; here: the memoization of lazy recursive calls.

The program below generates reasonably fast[5] hundreds of terms, and finally breaks because of overflows, since the series are divergent (30 terms yield numbers of the order of 10^{45}. We can rescale $\hat{x}$, but no factor const^n can prevent the explosion. We could use Euler or other resumming techniques, whose lazy implementation can be found in [20], but we shall not do this here).

```
psi k | k>=0 = lpsi!!(fromInteger k)
```

[5]some seconds; after 100 terms on a popular platform (a PC) the delays become visible…

```
     | otherwise = 0
lpsi = 0 : ps 1 where
 ps k = ((elmat k 0 :>
  sum[elmat k m * psi m |
     m<-[k-4,k-2 .. k+4]])/
        (negate(fromInteger k):>ep)):ps(k+1)
```

We have in a most straightforward way defined the infinite list `lpsi = [psi 0, psi 1, psi 2, ...]` where each `psi m` recurs indirectly, passing through the elements stored in `lpsi`. Note that the program remains co-recursive without any special cosmetics, `psi` *does not verify* whether the needed element has been already calculated, in order to retrieve its value!

Many other *simple and good* optimizations are possible. We could have tabularized the $\langle k|\hat{H}'|m\rangle$ matrix elements; we don't really need `ket`s, everything could be computed directly with `(N k)` and axes; the structure of $\hat{x}^4$ could have been simplified, we never need complex numbers, etc. But our point was that even without those conscious technical improvements, the formalism is perfectly usable on a very small computer.

3 Measuring

Typical theorists in computer science who "dare" to touch quantum problems, usually have no particular problems with the underlying algebra. There is, however, in our opinion, one weak point in several popular presentations and simulations (e.g., [21]) of quantum circuits and other aspects of quantum computing: the notion of *measurement* is often treated in a little *cavalier* way, usually sufficient for computing, and for the complexity analysis, but not always very good for the comprehension. In classical theories the measurement is an issue belonging to the domain of experiment, and to epistemology; a theorist may joyfully analyze the Turing or RAM machines, and re-use at will all the information specified by a given configuration (the state) of the machine.

In a quantum system any attempt to find out the information hidden in an unknown state will destroy it. 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"…

As we know, an unknown quantum state cannot be copied ("cloned") [22], so it is not possible to get around the difficulty imposed by the active role of measurement by producing the system duplicate, destroy it by the observation, and then do something more clever with the original.

The presented functional formalism attempts to comply with this restrictions, in the sense that we do not try to "cheat", to exploit any information to which we have no "legal" access, although classical computer programs have no intrinsic morality.

3.1 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 operator $\hat{A}$ in a state $|\psi\rangle$, which is denoted by $\langle\psi|\hat{A}|\psi\rangle$. In the example in the former section, (2.5), we computed the energy, $\langle n|\hat{N}|n\rangle$. One reads often 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:

$$|\langle\alpha|\psi\rangle|^2 = \langle\psi|\Big(|\alpha\rangle\langle\alpha|\Big)|\psi\rangle\,. \qquad (15)$$

In several papers devoted to the quantum computing this is the end of the story; the model gives us the probabilities, they are numbers, and we may stop here. If we decide to go further, the remaining part of the story 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.

No model can do more than that. This means that in order to get some results methodologically meaningful, we must repeat the simulated experience many times. Unless we are absolutely sure that the result of a quantum process is either a value "$\uparrow$" or "$\downarrow$", 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 numer generator many times, in order to gather a meaningful statistics. On the other hand, all serious algorithms in quantum computing are designed to generate a "settled", or "committed" states corresponding to classical configurations, and not to arbitrary superpositions thereof. In such a way *one* measurement should provide a definite (and definitive) answer.

Our package uses random number generators to produce *arbitrary* discrete distributions, and it works even in the case of infinitely dimensional bases, provided that the probability amplitudes vanish sufficiently fast. We know e.g., that the coherent state of an oscillator $|z\rangle$, where z is a complex number, which can be used to model a laser, gives the Poisson distribution for the excitation levels (or the number of quanta): $p_n = |\langle n|z\rangle|^2 = \mu^n/n!\exp(-\mu)$, where $\mu = |z|^2$, and n can be arbitrarily large. Yet, if the total energy of the system is limited, the average excitation level μ is not so big either, and standard techniques of generating Poisson distributions, e.g., [24] tell us how to measure our system.

3.2 Mixed states and density matrices

To be completed

4 Construction of Composite Systems

4.1 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 | O Oscil | ...
           | CP Qbase Qbase

instance Num s => Hbase Qbase s where
 bracket (Q x) (Q y) = bracket x y
 bracket (O x) (O y) = bracket x y
 -- ...
 bracket (CP x a) (CP y b) =
      bracket x y * bracket a b
 bracket _ _ = 0  -- Incompatible
```

In this case `axis` remains a linear operator, and it is possible to define the linear *tensor product* `(`><) of two axes by

```
infixl 7 <*>
(ax1 <*> ax2) (CP a b)
    = ax1 a * ax2 b      -- Forget it!
```

Of course, this requires that **ax1** and **ax2** are functions over **Qbase**, and not on individual bases for the qubit, the rotator, etc. This cascading tags may be a little clumsy, but this is not the worst problem: in fact, amalgamating the attributes of the subsystems within a composite data structure **CP ...** from the quantum point of view is a construction of the direct sum of the component Hilbert spaces, and has *no physical meaning*. It is a purely formal, artificial construct, without any *a priori* mathematical properties, but with too strong *structural* properties: in principle it is possible to disentangle a part of such a structure (one subystem) by an appropriate partial selector. In a quantum world this is impossible; this is the very essence of the Einstein-Rosen-Podolski paradox.

The main proposition in this section, considered 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 [23], but the book [15] (and many, many others) provides a complete discussion of the related mathematics. The formulae above, defining **Qbase** and the product of axes *will not be used at all!*, and we start from the beginning. We leave the axes as they have been defined in the section (2.1).

If a ket is a linear function defined on axes, a tensor product of two (or more) kets is a bi-linear (multi-linear) functions 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.

In general, 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 need seriously the multi-parametric classes with functional dependencies in order to be suficiently 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 the obvious, 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 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 **aket = ket B0 <*> ket B1**, and use it in our calculations. It is possible to prove that the tensor product is associative, although obviously non-commutative. In mathematical notation instead of $|\psi\rangle \otimes |\phi\rangle$ we will write simply $|\psi\rangle|\phi\rangle$, 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 trivially one subsystem, although by performing a partial measurement (applying the vector to an incomplete set of Hbase arguments), the arity of the state is reduced. But the result is (usually) not normalized, and needs thus some re-interpretation. 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 constitute a whole, even if the two subsystems are separated in space by a large distance. This is a conceptual problem which has been discussed thousands times, we shall not pursue this topic, we want only to signal that a simulator of a quantum system cannot be modularized into small, local units, each dealing with a small local sector of the global state.

This is true for *any* computer model of a quantum system. Does the functional programming have any advantages wrt. modelling approaches which use bit strings and complex arrays? Our answer is: yes. The laziness permits to keep relatively large tensor products in a semi-developed form, and facilitates the implementation of Bennet tricks [25] which reverse the computation flow in order to get rid of the auxiliary garbage. These investigations will be presented in a forthcoming work.

4.2 Dual tensors

This section is very short. First, if we want to construct *elementary* two- (or more, but practically restricted to few) sub-systems, say, $|0\rangle|1\rangle$, we don't need to apply explicitly the tensor product of single kets, we may start with multilinear primitives, e.g.,

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

Passing from such kets, or from any combinations thereof to axes is trivial, the answer is

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

and we see that a compound axis is also a bilinear function, and doesn't involve any "classical" Cartesian product of the associated Hbase labels. The norm of such a (ket) vector for the **Qubit** system is given thus by

```
norm2_p ktp = axnorm2_p (dual_p kt2)
where
 axnorm2_p ax2
   = abs2(ax2 B0 B0) + abs2(ax2 B0 B1)
   + abs2(ax2 B1 B0) + abs2(ax2 B1 B1)
```

4.3 Operators on tensor product states

We shall define now the tensor product of operators. 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:

$$\hat{A}_1 \otimes \hat{A}_2 \Big(|\psi_1\rangle \otimes |\psi_2\rangle \Big) = (\hat{A}_1|\psi_1\rangle) \otimes (\hat{A}_2|\psi_2\rangle) \qquad (16)$$

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.

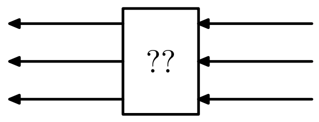

Figure 4: Operator on a composite state

In different words: we have a set of "input", and a set of "output lines", like on Fig. 4, and we have to construct *one* object which performs this transformation. It is interesting to observe that when 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 similar to the CPS programming, which — as we know — is able to deal with multiple arguments-to-multiple results functions.

Constructing the product of two operators is straightforward:

```
boost2 ap1 ap2 ktp  =
  \ax1 ax2 -> ktp (ap1 ax1) (ap2 ax2)
```

where **ap1** and **ap2** are operators acting on single axes, and **ktp** is a 2-ket. Such factorized object can be depicted as on Fig. 5.

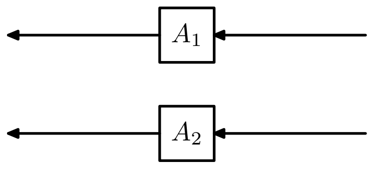

Figure 5: Tensor product of operators

The recursive construction of N-argument operators may follow the recipe similar to this used already for kets, but now we are in a different situation: the arguments of our operators are linear functionals themselves. The construction of the tensor product (acting on kets) of two linear (one-argument) operators (acting on axes) is easy, and shown above. Suppose now that we have two operators acting on kets, one of them is elementary, and the other — multilinear:

```
op kt = \x -> kt (ap x)

opm ktm = \y1 y2 ... ym ->
      ktm (bp1 y) (bp2 y) ... (bpm y)
```

(where lack of indices on **y** in **(bpk y)** suggests that the corresponding operator may depend effectively on many arguments; it is not necessarily a tensor product).

The product **opp = op <*> opm** will have as its specification the following pattern:

```
opp ktp = \x y1 ... ym ->
      ktp (ap x) (bp1 y) ... (bpm y)
```

This can be reduced to:

```
opp ktp = \x -> opm (ktp (ap x))
        = op (opm . ktp)
```

or, simply: **opp = (op .) (opm .)**. Unfortunately, when the left argument of the tensor product is multilinear itself, the formula gets more complicated: **op2 <*> opm = (op2 .) ((opm .) .)**, **op3 <*> opm = (op3 .) (((opm .) .) .)**, etc. We have to construct a recursive generator for such types in a general case.

To be completed. Stuck...

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. Good, we know already how to make tensor products, and we know that the operators form a vector space, so we can combine them linearly.

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 trivialization of the problem, of course there are legal quantum processes which create or annihilate particles, everything depends on the internal structure of these "lines". For very simple systems, such as qubits realized as flipping spins $1/2$, the statement is true because of the conservation laws.

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., [4]) 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. (6) is called the "controlled-NOT" gate, corresponds to the transition: $|x\rangle|y\rangle \to |x\rangle|x \oplus y\rangle$, and has

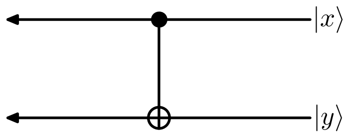

Figure 6: Controlled-not gate

the following definition:

```
cnot kt x y = p B0 + p B1 where
   p b = kt (qproj x b) (xor (axis b) y)
   xor r = r B0 *> id <+> r B1 *> qnot
   qproj x b = x b *> axis b
```

Notice that the gate performs a measurement (filtering), since it splits the state explicitly into two projections. It is not possible to avoid this. In the next section, (5.2) we add some comment to it.

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 aparticular 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 our simulated model obviously we won't obtain anything miraculous.

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

$$|x\rangle|y\rangle \to |x\rangle|f(x)\oplus y\rangle \tag{17}$$

```
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. (7). Two

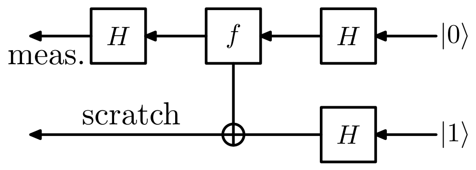

Figure 7: Deutsch problem

assigned 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

$$\begin{aligned} |0\rangle|1\rangle &\to \frac{1}{2}\left(|0\rangle+|1\rangle\right)\left(|0\rangle-|1\rangle\right) \qquad (18)\\ &= \frac{1}{2}\left(|0;0\rangle-|0;1\rangle+|1;0\rangle-|1;1\rangle\right). \qquad (19)\end{aligned}$$

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

$$\begin{aligned} &\to \frac{1}{2}\left(|0;0\rangle-|0;1\rangle+|1;0\rangle-|1;1\rangle\right)\\ &= \frac{1}{2}\left(|0\rangle+|1\rangle\right)\left(|0\rangle-|1\rangle\right), \qquad (20)\end{aligned}$$

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

$$\begin{aligned} &\to \frac{1}{2}\left(|0;0\rangle-|0;1\rangle+|1;1\rangle-|1;0\rangle\right)\\ &= \frac{1}{2}\left(|0\rangle-|1\rangle\right)\left(|0\rangle-|1\rangle\right). \qquad (21)\end{aligned}$$

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 — $|0\rangle$.

We can show the coding, but first a few words about the nonchalance of this derivation. What kind of mathematical object represents $f(x)$ in (17)? Is it a state description, suggested by its presence in a ket? Or a number 0 or 1, used numerically? Remarkably, in several introductory articles this problem is never explicited, the authors put or extract numbers into, or out of kets without any comments. It is possible, because they didn't try to implement states in an *abstract* way, as we did. Actually the function f ***cannot*** be an operator on general quantum states, it can do something only to a classical configuration, not to a superposition.

In our framework the situation is absolutely clean, **`f :: Qubit -> Qubit`**. We define two such objects:

```
fmut = id
fcst = const B0
```

and the circuit is represented by the following construction:

```
in1 = (had <*> had) (ket B0 <*> ket B1)

had_a = had <*> id
xout = had_a (fcnot fmut in1)
yout = had_a (fcnot fcst in1)
```

It suffices to measure those last states 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 state is proportional either to $|0\rangle$ or to $|1\rangle$.

In this introductory paper we cannot show more elaborate examples, nor show how the laziness helps to deal with long qubit sequences, but we believe that the overall flavour of the proposed framework is already sufficiently visible.

6 Conlusions

... in a nutshell: It is difficult to say when (if at all) we will have working quantum computers. We are nevertheless convinced that the paradigms of functional programming constitute a sound basis for their modelling, understanding, and also, in some possible future — their programming. The main purpose of this paper is to convince also the reader, by showing some simple, but not entirely trivial examples.

In this, preliminary work, we propose an abstract geometric framework permitting to define standard quantum entities such as states and observables, as functional objects, programmed in Haskell. The level of abstraction is so high that we can offer a common style for the simulation of very different quantum systems, and yet propose a set of effective algorithm implementations, permitting to obtain some non-trivial numerical results. Moreover, this genericity makes it more difficult to introduce errors in the program.

The mathematics used in standard quantum calculus is rather different from what one finds in a typical text on the theory of programming, so we have been annoyingly explicit in defining our vector bases, tensor products, etc. We believe, and we wanted to show that a modern, strongly typed and polymorphic functional language is actually the best tool for the implementation of those objects, although the Haskell type system is still not perfect for our purposes. This work will continue.

7 Acknowledgements

We thank Jan Skibiński for interesting discussions during the preliminary work on this subject.

References

[1] S. Brandt, H.-D. Dahmen, *Quantum Mechanics on the Personal Computer*, Springer, (1994).

[2] John Von Neumann, *Mathematical Foundations of Quantum Mechanics*, Princeton University Press, Princeton (1955).

[3] G.W. Mackey, *Mathematical Foundations of Quantum Mechanics*, Benjamin, New York (1963).

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

[5] 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).

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] D. Deutsch, *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proc. R. Soc. Lond. **A400**, (1985), p. 96.

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

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

[16] John Preskill, *Quantum Computing, pro and con*, Proc. Royal Society, Lond. **A 454**, (1998), pp. 469–486.

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

[18] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Higher-Order Symbolic Computations **14**, (2001), pp. 35–57.

[19] R.M. Corless, D.J. Jeffrey, M.B. Monagan, Pratibha, *Two Perturbation Calculations Using Large Expression Management*, J. Symb. Computation **23**, (1997), pp. 427–443.

[20] Jerzy Karczmarczuk, *Traitement paresseux et optimisation des suites numériques*, Proc. Journées Francophones des Langages Applicatifs, JFLA'00, (2000), pp. 17–30.

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

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

[23] 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.

[24] Donald E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).

[25] C. H. Bennet, IBM J. Res. Develop. **17**, (1973), p. 525. See also C. H. Bennet, SIAM J.Comput. **18**, (1989), p. 766.

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

A Multi-parametric classes

A 'class of types' in Haskell is a constraint, a relation fulfilled by the types which will be declared as its instances; the class says that its types-instances have some properties ensured by the existence of some polymorphic functions (class members). There is no conceptual obstacle that a class bind two or more types. Such multi-parametric classes are heavily used in our framework. First of all, it would be extremely rigid basing all the vector spaces involved, on the same type of scalars. Usually we use complexes, but for testing we can often forget about phases and use reals. In our semi-symbolic exercises we have used infinite power series, or infinite sequences belonging to a differential algebra, which permitted the implementation of the automatic differentiation algorithm.

So, we shall define more general vector spaces. Some elements remain, for example

```
class Vspace v
 where
  (<+>) :: v -> v -> v
  ...
```

needs no modification since no scalars are visible here. We have separated the multiplication by scalars to a different class

```
class Module v s
 where
  (*>)  :: s -> v -> v
```

where **v** is the type of vectors, and **s** denotes the type of the associated scalars. This class has instances in the scalar domains, where **(*>)** reduces to **(*)**, and the recursive, already mentioned in the text (2.3) clause permitting to lift the multiplication to the functional domain is:

```
instance (Module b s) => Module (a->b) s
 where
  (a *> f) x = a *> (f x)
```

A.1 Functional dependencies

Such type framework is too ambiguous. The Haskell type system forces us to declare practically everything, and often is not able to deduce the instance of the Module class. We used thus the fact that if the type which describes vectors is known, obviously its field of scalars is known as well. The augmented type system, which permits to the compiler the resolution of some ambiguities has been described in [26], and has been implemented in Glasgow Haskell.

The true **Module** definition is

```
class Module v s  | v->s
...
```

which means that the type **s** is uniquely determined by the type **v**. Our framework is much more polymorphic now. Unfortunately, although in principle it doesn't seem necessary, we were obliged to define the instances for each scalar field separately:

```
instance Module Double Double
 where
  x *> y = x * y
instance Module Cmplx Cmplx
 where
  x *> y = x * y
```

etc., since the attempt to declare the constrained general numerical instance

```
instance (Num s) => Module s s
 where
  x *> y = x * y
```

fails. For similar reasons which we cannot analyze, the tensor generic instance for numerical scalars:

```
instance (Num s,Module v s) => Tensor s v v
 where
  s <*> v = s *> v
```

doesn't work either. The concrete specifications:

```
instance (Module v Cmplx)=>Tensor Cmplx v v
 where
  s <*> v = s *> v
```

and for other scalars: **Double**, **Series Cmplx** etc., work reasonably well. It seems thus that our attempt to use Haskell in an abstract geometric context shows a usefulness of a possible strenghtening of its actual type system.

B Lazy Infinite Series

Here, and in the next section (C) we just show how these "semi-numeric", composite data structures are defined, and how to construct the arithmetics over them. The details are in [17].

Suppose that a pair $u = (u_0 \rhd \overline{u})$ denotes the infinite power series $u = u_0 + u_1 x + u_2 x^2 + \cdots$; obviously $\overline{u}$ is its tail $u_1 + u_2 x + \cdots$. The dummy (formal) variable x does not figure explicitly anywhere, and the Haskell name for $\rhd$ is **(:>)**. Structurally such a sequence is equivalent to a list $[u0, u_1, \ldots]$. Adding series is performed termwise by a generalized **zipWith** operation, which needs no comments.

The multiplication $w = uv$ of $u = (u_0 \rhd \overline{u})$ by $v = (v_0 \rhd \overline{v})$ is equal to:

$$w = (w_0 \rhd \overline{w}), \quad \textbf{where} \quad w_0 = u_0 v_0; \quad \overline{w} = u_0\overline{v} + \overline{u}v\,. \tag{22}$$

The division $w = u/v$, whose algorithm which uses indexed vectors, given e.g., in [24] is not so short, in a co-recursive formulation becomes extremely compact. From the identity $u = wv$ we see immediately the validity of

$$w_0 = u_0/v_0; \qquad \overline{w} = (\overline{u} - w_0\overline{v})\,/v\,. \tag{23}$$

Elementary functions, such as the exponential, pass through the differential identities (with x being the differentiation variable) fulfilled by such series. From $w = \exp(u)$ we deduce $w' = wu'$, and $w = w_0 + \int wu'$. But the differentiation of a series is just a multiplication term-wise of its tail by the sequence $\{1, 2, 3, \ldots\}$. The integration divides the argument by this sequence, but puts in front the new 0^{th} term — the integration constant. This makes the co-recursion possible, and this definition

```
serInteg c u = c :> snt 1 u where
 snt n (u0:>uq) =
  (u0/fromInteger n) :> snt (n+1) uq

exp u@(u0:>uq) = w where
  w = serInteg (exp u0) (serDiff u*w)
```

becomes effective. Our package contains procedures for the series composition, reversal, etc., but we will not need them here.

C Lazy Automatic Differentiation

This section contains a very small fragment of the paper [18]. We shall describe the lifting of numerical expressions within a program into a domain where a non-trivial *derivation* operation **df** is defined. All numerical expressions can be composed from elementary arithmetic operations (and some built-in functions with known properties). For simplicity we describe a 1-dimensional case, where there is a sense in saying that we have *one* "variable": it may be the argument of a function whose derivative we want to compute; it is identifiable in the program, but it has no specific name. Its main property is that its first derivative is equal to 1, and all higher derivatives vanish.

The program contains also some number of *constants* whose derivatives vanish. All "standard" expressions, say, e, are lifted to the domain of infinite sequences $e = e_0 \blacktriangleright \tilde{e}$, where e_0 is the "main value", the value of the original expression, and $\tilde{e} = e_1 \blacktriangleright e_2 \blacktriangleright e_3 \blacktriangleright \ldots$, with right-associative $\blacktriangleright$, represents the tower of all derivatives of e. In Haskell we define the following structure:

```
class Diff a where
 df :: a->a

instance Diff Double where
 df _ = 0.0
-- ...

data Dif a = Cst a | Dif a (Dif a)
instance Num a=>Diff (Dif a) where
 df (Cst _) = Cst 0
 df (Dif _ p) = p
```

where the variant **Cst x** is a natural optimization of — otherwise unavoidable — infinite chain **Dif x (Dif 0 (Dif 0 (Dif ...)))**. The expression **Dif e0 de** is the Haskell representation of $e_0 \blacktriangleright \tilde{e}$.

If the user writes a numerical procedure in which all the operations and functions are sufficiently polymorphic: not restricted to **Doubles** etc., but overloadable to, say, **Dif Double**, then it suffices to replace all implied constants c by **Cst c** (all explicit numeric constants are lifted automatically by specific **fromInteger** etc; converters), and the *variable* x by **dVar x = Dif x 1**. The **Dif** datatype is an instance of all needed numeric classes.

The lifted expressions are added or subtracted term by term, as in the case of series. The Leibniz identity which must be satisfied by a decent derivation operator results in the following recipe for the multiplication of $e = e_0 \blacktriangleright \tilde{e}$ by $f = f_0 \blacktriangleright \tilde{f}$:

$$e@(e_0 \blacktriangleright \tilde{e}) \cdot f@(f_0 \blacktriangleright \tilde{f}) = e_0 f_0 \blacktriangleright e\tilde{f} + \tilde{e}f \tag{24}$$

The division, and some elementary functions given below, are equally easy. (We omitted the trivial definitions involving the **Cst** sector.)

$$e@(e_0 \blacktriangleright \tilde{e})/f@(f_0 \blacktriangleright \tilde{f}) = e_0/f_0 \blacktriangleright (\tilde{e}/f - e\cdot\tilde{f}/(f\cdot f))\,, \tag{25}$$

and

$$\begin{aligned}
\exp(e@(e_0 \blacktriangleright \tilde{e})) &= r \quad \mathbf{where} \\
r &= \exp(e_0) \blacktriangleright \tilde{e} \cdot r\,, && (26) \\
\sqrt{e_0 \blacktriangleright \tilde{e}} &= p \quad \mathbf{where} \\
p &= \sqrt{e_0} \blacktriangleright (0.5 \cdot \tilde{e})/p\,, && (27) \\
\mathrm{atan}(e@(e_0 \blacktriangleright \tilde{e})) &= \mathrm{atan}(e_0) \blacktriangleright \tilde{e}/(1 + e \cdot e)\,, && (28)
\end{aligned}$$

etc. All expressions belong thus to the lifted domain, and in order to retrieve their main values, we apply the function **`dVal (Dif x _) = x`**. All derivatives are available “for free”. As we have seen in the example in section (2.5), sometimes we don’t need the values of the derivatives in the final output, but we use them during the calculations, e.g., to solve differential recurrences. The computations in quantum mechanics use them very frequently.