# $\mu$, A Minimal Patch for Quantum Computation, Information and Foundation

Sajad

October 13, 2020

**Abstract**

Today, as quantum computing is trending more and more, while companies and universities are making different tools for different purposes, the lack of a general-purpose, well-designed, and futuristic programming language is felt. Here is a try to design a patch to be added to (almost) any programming language to make it support quantum algorithms and simulations. Moreover, this definition may help to represent clearer not only quantum algorithms but also many other problems in quantum. Although the design process was inspired by theoretical computer science concepts, specifically PL concepts, it tries to be intuitive and friendly for mathematicians and physicists. The process of design, presented specifications and details of implementation is discussed in this paper. While the real examples are available at Github.

# Contents

# 1 Introduction

This text, assumes you have proper knowledge in quantum computing. If you don't, you can read [**neilson**] or any other standard books.

# 2 Purpose

- Easier way to write, read and understand QC algorithms

  The way we write down quantum algorithms is seemed to be not a projection of what we understand from them. Often learning every single quantum algorithm need to be a task. That's maybe not just for its intrinsic hardness, and the way we often propose algorithms (mathematics for the definition of gates, and a circuit for its sequence) is disjointed and hard to follow. Maybe a more concrete way to represent makes them easier to understand, modify, and generalize.

  For that reason, we need to just an standard way to write down a psuedo that uniquely specifies the algorithm. But other purposes give us the reason why we need to implement such thing.

- Intuitive definition and computation for scientists

  People with mathematical backgrounds (mathematicians and physicists) are often having hard time learning a simulation/expermental tool, specially when it's designed by developers. this came from the different intuitions that these two groups have. A simulation/experimental tool must be at the same abstraction level that scientists are familiar with to be a comfortable. Another point is that, some may belive that modules and organizing functionalities in group are not compatible with scientific idea of solving problem, it may be correct, but it's clear that ability of importing and having all tools together, is not necessarily logically in contract with organizing them as groups at the first place.

  It's not necessary to point that using the scientific terms, operators, and functions are the baseline to do to make it scientist-friendly.

- a Swiss army knife, not a giant supertool

  To cover all of the wide-range applications of a language describing and simulating QC/QI processes, it doesn't need to be giant set of tool. As it's been seen in the computer science, general-purpose programming languages are not that much big, in terms of core functionalities. A limited set of tools tend to be more memorizable and more enjoyable for developers.

- Layered and extensible patch

In continue to the previous feature, software architecture can now helps us, designing easily extensible tools. The design pattern that it's good to be followed is layered. The reason behind that is that in the pure science we saw, QC, is represented on a circuit that is made over the quantum postulates, over a linear algebra.

- Futuristic

  Maybe it's too early to judge about the future of quantum computing tools, but it doesn't mean that we shouldn't step forward to make it.

# 3 Design

## 3.1 Basic Hilbert Space Interface (BHSF)

Before gettin into the main subject, we offer a library with a common set of actions on Hilbert space. This is like the effort people made before to design BLAS (Basic linear algebra system), defining fundamental tools that are necessary for those kind of computations.

Structurally, BHSF is guessed defined be over a BLAS. Here we just define the interface in term of types, functions and a brief structure (which is technically implmentation, not definition)

**Types:**

- `H` is the type that represents a single Hilbert space (tensor-product spaces are not counted in).
- `Vector` which represents a vector in a set/list of Hilbert spaces.

**Constructors:**

- `H(dim: Int)` $\to$ `H`: For constructing `H` from its dimension.
- `Vector(v: List[Complex], H)` $\to$ `Vector`: This function constructs a vector from a list of coefficents each corresponds to each basis of `H`, it's like $\sum_i v_i \left|i\right\rangle$

**Conversions:**

- `Vector` $\leftrightarrow$ `Complex`: vectors `v` with `space(v)` $=$ `[]`, are isomorphic to complex numbers.

**Functions:**

- `dual(H)` $\to$ `H`: This function, returns dual of the space. This is useful because an operator on `H` is like a vector in the `dual(H)` $\otimes$ `H`.

So we ask this function to satisfy `dual(H)` $\neq$ `H` and `dual(dual(H))` $\neq$ `H`.[1]

- `idual(H)` $\rightarrow$ `H`: inverse of the `dual`.
- `dim(H)` $\rightarrow$ `Int`: Returns dimension of the `H`.
- `space(Vector)` $\rightarrow$ `List[H]`: This function returns list of Hilbert spaces that this vector belongs to tensor product of them. This is just used to discuss other functions.[2]
- `dual(Vector)` $\rightarrow$ `Vector`: This function returns dual of the vector.
- `idual(Vector)` $\rightarrow$ `Vector`: This function returns inverse dual of the vector.
- `pnorm(Vector, p: Real)` $\rightarrow$ `Real`: This function returns $\ell^p$-norm of the vector.
- `tr(Vector, List[H])` $\rightarrow$ `Vector`:
- `morph(Vector, from: List[H], to: List[H])` $\rightarrow$ `Vector`: Changes space of the vector. Hilbert spaces in `from` are replaced with `to`, with simplest isomorphism between their basis. (`from` and `to` must have the same dimension)

**Operators:**

- `Vector * Vector` $\rightarrow$ `Vector`: Tensor contraction similar to dirac notation. It means that for `a, b: Vector`, If `space(a)` = `[dual(H)]` and `space(b)` = `[H]` then `space(a * b)` = `[]` and `space(b * a)` = `[dual(H), H]`. And obviously if `space(a)` = `[`$\mathcal{H}_1$`]` and `space(b)` = `[`$\mathcal{H}_2$`]` then `space(a * b)` = `space(a * b)` = `[`$\mathcal{H}_1$`, `$\mathcal{H}_2$`]`.
- `Vector + Vector` $\rightarrow$ `Vector`: addition of two vector.
- `Vector - Vector` $\rightarrow$ `Vector`: subtraction of two vector.
- `Vector * Complex` $\rightarrow$ `Vector`
- `Complex * Vector` $\rightarrow$ `Vector`
- `Vector / Complex` $\rightarrow$ `Vector`: scalar multiplication and division.

[1] We know that mathematically $H^{**}$ double dual of a space is not equal but isomorphic to itself. This fact is useful in the implementaton of a general library to support channels and superoperators with a simple dirac-like notation.

[2] Although we ignore the isomorphism between $H^{**}$ and $H$, we use this fact that $H_A \otimes H_B$ is ismorphic to $H_B \otimes H_A$ and we assume the isomporphism is identity. therefore `space` function will not guarantee the order.

**Example**

The following psuedo-code shows the functionality of BHSF.

```
h = H(3)
v1 = Vector([1,0,0], h)
v2 = Vector([0,1,0], h)

dual(v1) * v2) ; same as <v1|v2>
v1 * dual(v2)  ; same as |v1> <v2|
```

## 3.2 Typed Quantum Computing

We often use qubits for the purpose of quantum computation, while the bit itself is not used in classical computation. It's mainly used in hardware description languages and not programming languages and algorithms.

Types, such as `Int`, `Int8`, `Bool` are the core things that we use to think about algorithms. On the other hand, in quantum computing we are also thinking in terms of groups such and fields (for the basis) which are apperently similar.

The reason behind working with types is pretty obvious, but the main challenge is to define proper types and our map current tools, qubits, sites and gates well in terms of types.

### 3.2.1 Types and Variables

In a pretty simple quantum algorithm, we may use multiple qubits, means that we have a vector (called the system state) that belongs to a $\mathcal{H}^n$. then we say we're applying a gate on $i$-th qubit, we are refering one of those Hilbert spaces, then defining a linear operation on it and applying it to the system state.

Therefore, qubit is somehow represented via a Hilbert space, while the system state is globally specified.

We define a parameteric type `Q`, `Q<T>` (C/C++/Java) `Q[T]` (Scala) `Q{T}` (Julia), each instance of it, is basically a Hilbert space with the basis that are all possible values for type `T`.

Our system state is also something that belongs to all of the defined `Q` variables.

When we're defining a new `Q` which is like adding a new qubit/qudit/quantum subsystem to our system, we must set its initial value, therefore ve define type `Val{T}`. Each value of type `Val{T}` is a vector with basis that are `T` values. Therefore we can create a `Q{T}` with initial value `Val{T}`, then as a result the system state will be now the tensor product of the old system state and the initial value, and `Q{T}` will be referring to the added Hilbert space.

### 3.2.2 Composite Types

### 3.2.3 View Types

### 3.2.4 `Q`-type

While physicists/mathematicians think about groups, maps and measures, within different levels of abstraction, and computer scientists think about primitives, structs, datastructures and more, the major portion of quantum computing tools are offering "Register Arrays" as the main tool for computation, which is disappointing for anyone other digital systems experts.

### 3.2.5 Views

Assuming `a: Q{T}`, `b: Q{V}` a type, `(a, b): Tuple{Q{T}, Q{V}}` can be intrepreted as `Q{Tuple{T, V}}`, this is called viewing, or more specifically, cross view.

Also, for a compound (cartesian-cross) type, such as static-length arrays, tuples, structs, a projection can have a similiar viewing called projection view.

A more quantum view, is to use another basis (the same or of another type, but with the same cardinality) as another representation for a vector, an extreme example is spins in different axes or total spin basis.

### 3.2.6 Multiview Type

When we have a quantum hilbert space that we can assign different basis to it, while we shall not promote one of them as the main, we can use union, an old abondened tool in programming. But if you're not familiar with it, we just use a compound structure, assigning different basis views to its fields.

### 3.2.7 Mutability

This are not mutable, but this is not bad, non-copying principle is not compatible with copying in closure and more in immutable functional programming.

but in another hand we can present another meaning for mutability / immutability, views can be read-only or read-write, those we can set them mutable / immutable.

### 3.2.8 Move Semantic

### 3.2.9 Operators and Superoperators

Just by using match-case and some automatically applied castings, it can be intuitively written.

### 3.3 Code as Operators

As a beloved target, codes in the operators that are enslaved within lambda expression, may can be freely written.

## 4 Implementation

### 4.1 Julia

## 5 Examples

### 5.1 Algorithms

#### 5.1.1 BV

```
def BV(func, n)
    inputs = [var(0, type=bool) for _ in range(n)]
    result = var(false)

    for i in inputs:
        i.apply!(u(lambda x: ket(0) + (-1) ** x * ket(1)))

    result = t(result, inputs).apply(u(lambda (x, y) : y + func(x))

    return result.measure()

gate<n> fourier = match n
    case 0:
    otherwise: fourier<n - 1>

optional<T> grover(f: pure<bool, T>, init: set<T>, real p) {
    q<T> vec = init;

    gate reflect = x: T => match x
        f(x) ==1, 1
                  -1

    vec -> (reflect f)^(1/p)
    T res = vec -> measure
    if f(res)
        return Some(res)
    else
        return None
}
```

```
real QAOA<T> problem(f: pure<bit[], real>, int level) {
    q<T> vec;
    hamiltonian C = t : T => f(t)
    hamiltonian B = X[n]
    vec -> exp(iC) -> exp(iB)
}

real annealing(f: problem,
```