μ, A Minimal Patch for Quantum Computation, Information and Foundation

Sajad

September 27, 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, about an introductory course such as [Nilson and Cheng book] or [Vahid lecturenotes...].

2 Purpose

- Easier way to write, read and understand QC algorithms

  The way we write down quantum algorithms felt 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.

- Intuitive definition and computation for scientists **NOT COMPLETED** First, addressing the needs of scientists (mathematicians and physicists) to simulate and compute, Then just a standard form to describe things in a quantum manner.

- a Swiss army knife, not a giant supertool **NOT COMPLETED** A minimal and complete set of tools,

  Fully understandable for developers, engineers and scientists.

- Layered and extensible patch

- futuristic

3 Design

3.1 Basic Hilbert Space Interface (BHSF)

Before getting into the patch, a library of common actions on Hilbert spaces is necessary.

Structurally, BHSF is guessed defined be over a BLAS.

We define interfaces as below

Assumptions: `int` is the type of int and `...int` is the type of a sequence of int. Also `T{args}` is a generic type, with respect to `args` that are parameters that can be both types and values (like Julia generics and C++ templates).

Types:

- `H` is the type that represents a unique Hilbert space.
- `Ket{dims: ...int}` which represents a vector in the Hilbert space of $\mathcal{H} := \bigotimes_{i=1}^{n} \mathcal{H}_i$ where $\dim \mathcal{H}_i =$ `dims[i]`
- `Bra{dims: ...int}` which represents a vector in dual space of $\mathcal{H}$ (a.k.a. $\mathcal{H}^*$)
- `Op{dims: ...int}` which represents a linear operator on $\mathcal{H}$

Surely, this generic parameters are not the necessary part of this design, as it's not implementable in many languages. They are mention here to present a better (and more restricted) structure at the first galance and can be easily ignored in the implementation.

Functions / Operators:

- `*(a: Hilbert, b: Hilbert): Hilbert` the result is same as tupling Hilbert spaces together
- `'(h: Hilbert): Hilbert` is dual space

```
Functions:
- ket{dim: int}(i: int) -> Ket{dim}
- bra{dim: int}(i: int) -> Ket{dim}
- conj(k: Ket{seq}) -> Bra{seq}
- conj(b: Bra{seq}) -> Ket{seq}
- conj(o: Op{seq}) -> Op{seq}
- permute{perm: ...int}(k: Ket{seq}) -> Ket{seq[perm[0]], seq[perm[1]]

Operators:
s : Complex , k : Ket{seq}, b: Bra{seq} o: Op{seq}
s * k -> Ket{seq}
    |-> a * b = Bra{seq}

    |-> b * k : Complex
    |-> k * b : Op{seq}
    |-> b * o : Bra{seq}
    |-> o * k : Ket{seq}
```

```
k1, k2: Ket{...}
    |-> k1 + k2 : Ket{...}

k1: Ket{s1}, k2: Ket{s2}
    |-> k1 o k2 : Ket{s1 . s2}
```

with the following rules

3.2 Core

By adding a special generic type called Q, to any modern programming language we can make it possible.

`Q<T>` (C/C++/Java) `Q[T]` (Scala) `Q{T}` (Julia) means a quantum space in $\mathcal{H}(T)$ calling Hilbert space with T as basis. But we know that having multiple `Q`-typed values, makes our Hilbert space much larger that is kronoker product (tensor product) of spaces/vectors.

3.2.1 `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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 Move Semantic

3.2.6 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

```
// pure function
bool BV<n>(f: pure(Z<2>, Z<2>^<n>)) {
    q<bit[n]> vec; // somehow set to zeros
    q<bit> res;
    vec.each(x => x -> H);

    gate oracle = (x, y) => y + f(x);
    // then gate become a H<Z2^n> x H<Z2> operator
    (vec, res) -> oracle

    return res -> 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,
```