Group10 Final Report

Denis Tra Bi, Tianheng Xiang, Yishan Zhao

May 17, 2021

1. Topic and Motivation

During this semester, we are going to explore the topic of quantum lambda calculus. To be specific, we will study the idea of quantum lambda calculus and relevant topics, and then invent a new functional quantum programming language base on that idea. Although lambda calculus was developed in the 1930s, the quantum lambda calculus is still a quite novel field yet to be exploited.

Current quantum programs generally consist of two distinct and inseparable parts, the quantum algorithm that operates exclusively on qubits and quantum gates and driver written in conventional programming languages. The driver provides a window for input and out of the quantum black box, while the quantum programming language describes the algorithm at a low level, almost as if it is the assembly code in the quantum world. *Quantum Lambda Calculus* introduces a brand new perspective to the status quo. First, it defines a brand new interaction between classical data and qubits. For example, the pattern matching block takes in a boolean variable and decides the specific operation on qubits based on the classical condition. Besides, quantum lambda calculus also benefits from its original classical form. Functional programming languages based on classical lambda calculus eliminates the awkward and tedious part of imperative programming languages, which allows developers to focus on the logic itself while maintaining expressiveness. Similarly, quantum lambda calculus provides an alternative approach to quantum computing: it is an abstraction of the current circuit model. It manipulates qubits at a higher level, utilizing the power of higher-order functions. The quantum functional programming languages are implementations of quantum lambda calculus, and thus we would like to study these languages since they inherit the merits of quantum lambda calculus.

After acknowledging the status quo, we think a new functional quantum programming language is necessary. Not many quantum programming languages exist for the time being, and most of them are quite primitive and domain-specific. For example, Quipper, which we will discuss in detail in Section 2, is a domain-specific language that works with Haskell. Although it is categorized as a functional programming language, the unitary operations are mostly imperative. That is, qubit is a mutable data type. Apart from that, QCL, quantum computation language is also considered to be a functional programming language. However, it has just the same problem with Quipper: qubit operations are actually imperative. Moreover, we discovered that programmers can actually "clone" a qubit state by creating an entangled pair with identical states. Although it is a way to bypass the restriction of no -cloning theorem and make the code easier to write, we don't think it is a preferable approach because this is actually very different from the meaning of cloning. In addition to that, all of the existing languages don't support quantum higher-order functions, further hampering the expressiveness of functional programming.

It is also crucial to code quantum programs in a high-level language. As mentioned above, many quantum programming languages are extremely low-level that they basically build up quantum circuits, just like assembly languages in the quantum world, even programmers themselves could easily get lost. Apparently, this is far from ideal for developing any complicated algorithms. For comparison, many simple algorithms like Dijkstra's algorithm or quick sort that runs on classical computers require some sort of flow control and data structures, and they are especially easy to implement in functional programming languages. It is reasonable to think that similar scenario can happen on quantum algorithms as well. In other words, such functional quantum programming language would have been a great help to computer scientists. Apparently, functional quantum programming is the future.

2. Summary of relevant literature

After conducting rounds of research, we found out that Peter Selinger, who is also the author of a document provided in the project topics, offers some perceptive insights on this topic. He also seems to be the primary contributor to quantum lambda calculus theory. In the publication *Quantum Lambda Calculus*, he described the topic in great detail, including a sophisticated quantum-based typing system and structures of various operators. The type system, which is discussed in great detail in chapter 1.3 and the following ones, can be invaluable to our project. For example, the typing rules and the rigorous definitions of type inference are illustrated using diagrams and formal proofs. With the power of type system, there will be an extra layer of bug-proof mechanism in our language. However, due to the complexity of implementing such algorithm [6], we do not consider integrating such a system into the interpreter.

During the research, we discovered **LIQ*Ui*|⟩** (Language-Integrated Quantum Operations), a quantum library for F# language. The user manual is well organized and we thought it might be a good reference for our projects in practice. However, it proved to be inadequate for our project: its primary focus is on generating quantum gates, drawing circuit diagram, and quantum program verification. Unfortunately, none of these are in our exact need.

We also found the book *Semantic Techniques in Quantum Computation* to be quite relevant to our topic. The authors offer a more detailed explanation on Valiron and Selinger's paper[3]. On the other hand, this book has much overlap with *Foundations of Quantum Programming*. Through scrutiny, we found this book focuses on the structure of quantum languages in general. The author proposes some valuable concepts such as quantum recursion and quantum "while" language. Most of the chapters are not closely related to functional programming, except for quantum recursion, and thus, we primarily focused on the chapter of quantum recursion.

The author of the book *Quantum Lambda Calculus*, Peter Selinger, is also the inventor of the language *Quipper*. We thus conducted research on both his papers on *Quipper* and the official open source repository. Later we found out that, instead of being an independent programming language, it is more similar to a DSL that relies on the infrastructure of Haskell [4]. There are examples and simple demonstrations on their website and in paper [4], but since we have no knowledge on Haskell, it proved to be challenge to reproduce the outcomes of the codes. Therefore, we tried to understand the high level design philosophy instead, although neither was there any helpful insights. We also looked at other quantum functional programming languages such as Qurry [5], but all of them are mere proof-of-concepts.

3. Goal of the project

Since we are computer science majors, we agreed that a CS-like approach would be more appropriate to us. Therefore, we narrowed the topic down to implementing a basic and minimalistic quantum lambda calculus interpreter for a programming language that could handle both classical and qubit data types.

- solutions to the no-cloning theorem
- extending the classical concepts into the quantum context

These two aspects, together with some other essential and worth-mentioning auxiliary features, will be further explained and discussed in the following subsections.

3.1. The Language

Our new quantum functional programming language will be called *QAL*, standing for Quantum Abstract Language. We use F# to implement the interpreter, which consists of the following parts: a lexer, a parser, and most importantly, an interpreting module. The lexer and parser are basically the same as in classical interpreters, whereas the interpreting module incorporates a few original ideas. We use Q# to handle the actual unitary transformations and measurements on qubits.

F# and Q# are two languages that run on Microsoft .NET Core framework and requires .NET5 as their standard library. We intentionally chose this configuration because it is extremely easy for these two languages to communicate with each other as they run on the same framework and shares the same runtime environment. However, there are also some limitations, especially for Q#. According to the official documentation, Q# must run on CPU architectures that support AVX instruction set, which is virtually exclusive to Intel CPUs.

We decide to design *QAL* in the simplest form, which means *QAL* will not have fancy syntactic sugars supports which commonly exist in many other functional programming languages. The unsupported features are as follows.

- Anonymous function: Users are required to have their own defined functions with names declared.
- Partial application: Users must use a wrapper function to apply some of the arguments
- Boolean Type: Instead, we will have '0' as 'false' and '1' as 'true'.
- Type system: Data types still exist, while explicit declaration of a variable as another type and user-defined types will not be supported. There are no type-checks in function signature and its argument(s).
- Pattern matching: Only a subset of ML style pattern matching is supported. For example, there is no support for recursive data structures such as nested tuples or 2-dimensional arrays.

Our syntax is rigorously defined as follow:

- E is the set of all valid expressions, and $E_1, E_2, \ldots, E_k \in E, k \in \mathbb{Z}^+$.

- M represents match blocks, which are in the form of "match ... with (Cases)".
- $Case$ is a single match case, consists of a *pattern*, a right arrow and a branch expression.
- L is a let block which can be used to declare variables and functions.
- V is a general union of all values such as integers, complex numbers, etc.

$$
\begin{aligned}
E &::= (E_1) \mid M \mid L \mid (E_1\ E_2\ \ldots\ E_k)^1 \mid V \mid (E_1, E_2, \ldots) \\
M &::= \text{match } E_1 \text{ with } Case* \\
Case &::= pattern \rightarrow E \\
L &::= \text{let } var\_name = E \text{ in} \mid \text{let } fun\_name\ param* = E \text{ in} \\
V &::= \mathbb{Z} \mid \mathbb{C} \mid Qubit \mid Array \mid Composite\_System
\end{aligned}
$$

3.2. Design of QAL

The QAL language has similar syntax with OCaml and supports various common features such as let binding, pattern matching, and user-defined functions. OCaml is a typical functional programming language that all our group members familiar with. Racket, a functional PL member of LISP family, was also a choice that we would like to adopt. The reason why we did not choose Racket is because of its design of syntax in declaring variables and functions.

In Racket, declaring a variable and a function requires two distinct syntax structures: let binding block for variables and "define" block for functions. While the former is relatively easy to implement, we need to further generalize the structure of AST to facilitate the "define" block. For example, some expressions can have an arbitrary number of "define" blocks as their nodes, while others cannot have them at all. This extra generalization forces us to invest more time and effort into these details that are essentially irrelevant to quantum computing and we think it is not worthwhile to do so.

Since QAL inherits the syntax and style of OCaml, it supports features such as let binding, pattern matching and user-defined functions. Since QAL is a quantum programming language, it also supports quantum data and operations, such as qubit, qubits in tuples, and composite systems. QAL also has quantum version on some important functional programming languages features such as closure, recursion, and higher-order functions, we will elaborate this feature in Section 3.3

One important aspect of being a quantum programming language is that, QAL use enforce the no-cloning theorem so that no quantum type varible could be used twice after declared, in this way we adopt linearity in QAL and will be discussed in detail in Section 3.4

3.3. Quantum Closure, Quantum Higher-Order Function and Quantum Recursion

Closure and recursion are essential to classical functional programming languages, and we include these features in a quantum manner in QAL. In addition, we kept the concept of higher-order function which makes user-defined functions easier to write and use.

[1] Apply $E_2 \ldots E_k$ to the return function of E_1, here E_1 returns a function.

3.3.1. Quantum Closure

Classical closure allows any user-defined function to capture variables that already reside in the environment before the declaration of those functions, so that local variables inside functions can later refer to the environment, when the function is called.

Initially, we implemented this by packing the environment before the function declaration, just like in other normal programming languages. However, this conventional approach doesn't work well with linearity. We realized that synchronizing the original environment and the function's closure is quite problematic due to the limitations of classical functional programming language.

Thus, we came up with the solution to have the environment be a partially pass-by-value and partially pass-by-reference list, which is exactly why we define our environment to be a tuple with a mutable boolean reference. No matter which environment a quantum data structure is in, its Boolean reference will always change accordingly to all the environments that contain them. We will elaborate this approach in Quantum resource management section.

In this way, we have our version of quantum closure that obeys no-cloning theorem.

```
let g =
  let q = (new 0) in
  let f x =
    (add (measure q) x)
  in
  f
in

(g 0)
```

Listing 1: Closure Example

The code present in Closure Example is a simple example of how we apply quantum closure. The variable q that is declared before the function f could be used in the function, and therefore it is contained in the closure of function. Here, g is actually an alias for f. Later, when f is called, the operation of measuring the qubit q will look for the local variable q inside the quantum closure. The final result would then be 0 of integer value in this case.

```
Integer_Val 0
```

Result of Closure Example

3.3.2. Quantum Higher-order Function and Quantum Recursion

Higher-order function is a special category of functions that either accepts functions as parameters or returns functions.

For instance, here are some frequently-used higher-order functions in some classical functional programming languages that also exist in *QAL* with their semantics slightly changed:

- map: The classical "map" function iterates through all elements in a list and performs the specified function to these elements: as the name suggested, it is basically a mapping from one set to another. However, when it is applied to a collection of qubits, the original collection is virtually inaccessible afterward.

- fold: The fold function will take a function, a list, and an accumulator as parameters. The fold function will recursively apply the input function in an iteration way to each

element and accumulate the result on the accumulator. It will eventually return the accumulator. Its quantum counterpart is essentially the same, but they are carefully designed to enforce no-cloning theorem.

A reason why we include higher-order function as a feature in QAL is because there is a specialized list in QAL that exclusively stores qubits, called "system", short for "composite system". In some classic algorithms that operate on an enormous number of qubits, they could be stored in such a list, then we can use either map or fold function to perform the required quantum operation. It is vital to any functional programming language since we prefer not to use iterative loops on the list.

The functions in QAL can also be treated as values to be passed in or out. This is extremely helpful in using higher-order functions like map and fold. For instance, a classical map function takes in a mapping and acts on every element to convert them to something else. Its semantic is similar in QAL, but it is under the constrain of the no-cloning theorem.

With the help of higher-order functions, our language can perform operations on an arbitrarily large number of qubits and keep the code human-readable at the same time.

But this feature would not be possible without implementing quantum recursion in the first place. Fortunately, since recursion is well-defined in classical functional programming, it is relatively easy for us to extend this concept to quantum computing: it behaves just as expected.

Here is an example of a quantum higher-order function.

```
1 let lst = (System 3) in
2   (map measure (map X lst))
```

Listing 2: Quantum Higher-order Function Example

Quantum Higher-order Function Example is a simple code demonstrating higher-order function map in QAL. This code first creates a system that stores several qubits, the keyword *System* is a standard-library constructor that will return a list (composite system) that could only contains qubit with given numbers of qubits in state zero. Here three qubits are initialized in the composite system, each qubit has state zero. Then the code uses a map function to apply X-gate on each of the qubits in the system, and finally measure the state of each qubits using another map function. After applying the X-gate, all the qubits in the system should have the state 1, thus the final result will be an array of three 1s.

```
Array_Val [Integer_Val 1; Integer_Val 1; Integer_Val 1]
```

Result of Quantum Higher-order Function Example

```
1 let apply_X_nTimes n qubit =
2   match n with
3   | 0 -> qubit
4   | _ -> (apply_X_nTimes (add n -1) (X qubit))
5   in
6
7   let q = (new 0) in
8   (measure (apply_X_nTimes 3 q))
```

Listing 3: Quantum Recursion Example

Quantum Recursion Example defines a recursive function that applying X gate on a qubit with given times. Here we apply the X gate on a qubit initially with state zero three times. The final result of this code would be one as we apply odd times of X gate on a qubit with state zero.

```
Integer_Val 1
```

Result of Quantum Recursion Example

3.4. Quantum resource management

Quantum data types in *QAL* will behave differently as to others. Because of the no-cloning theorem, we should not allow duplicated quantum variables and copying quantum qubits. In imperative quantum programming languages, this problem could be easily circumvented. For example, in *Q#*, every qubit will just evolve after several quantum operations: the referencing object (qubit) is changed while the reference is not. However, due to the fact that functional languages are designed to be pass-by-value and variables are essentially immutable, managing these quantum resources becomes a critical issue.

In some quantum functional programming languages, "cloning" is allowed. For example, *Quipper* allows a qubit to be "cloned" by returning an entangled counterpart with the exact same state as the original. Although this is a viable approach, we consider this to be a flaw in language design because it is not the correct semantic for "cloning".

However, in *QAL*, we will strictly enforce the non-cloning theorem and its underlying philosophy: once a qubit is created, it may not be cloned in any way. Moreover, after any quantum operation is applied to that qubit and causes any change in its state, this qubit may not be reused, and any subsequent quantum operation will be regarded as an invalid operation. This is named "linearity" in *On a measurement-free quantum lambda calculus with classical control* [1].

For instance, consider the following code snippet:

```
let qubit' = qubit in
```

Unlike classical data assignment, 'qubit' becomes inaccessible after this operation[2]. The reason is that we have to keep track of the references of qubit variables (pointers) so that no quantum variables may point to the same quantum resource. Otherwise, disastrous scenarios like this may happen:

```
let q = qubit in
  CNOT q qubit
```

The CNOT gate operates on two same qubits, which is unthinkable in quantum computing. Or something slightly less terrible:

```
let qubit' = X qubit in
let qubit'' = qubit in
...
```

Since "qubit" value has been changed since line 1, another access may get a different value than the original value of "qubit". This is especially unacceptable in functional programming since it implies implicit mutable variables.

Therefore we take the approach of linearity [1], which means no variables are allowed to use twice.

Note that this is quite different from what was described in [6] and *Quipper* language, because we think creating an entangled pair with the same state to imitate the "cloning" operation is not a good idea.

This leads to a profound shift in our language design.

We proposed two methods during investigation about dealing with the non-cloning problem, or two approaches to the linearity of our language: reference counting and reference tracking.

- Reference counting: There will be a list monitoring the current living qubits. When a qubit with a binding (reference) has been declared by the user, the binding name of that qubit will be added to that list. When there is a quantum function call that will make changes to qubits, before executing that function call, our interpreter will first lookup whether all the qubit arguments exist in that list, and raise an error if not. Then after the execution, all the qubit arguments will be deleted from the list to indicate that these qubits should not be used anymore unless such names are used later on to bind another brand new qubit.

- Reference tracking: A hash table will be used to keep track of the living qubits. The key will be the binding name of the qubit and value will be the id(hash value) of the key to represent the reference of that qubit. In other words, we give each qubit a unique reference to track the qubits. In this way, when there is a new temporary local binding to one existing qubit, it could be easily tracked by renaming the key of that qubit in the hash table.

At last, we adopt the reference tracking method with modifications. In our interpreter, we store local variables by maintaining an environment, which is a tuple list with the tuple consists of string, value and bool reference. Note that only the bool is a reference type. The strings store the variable name, values store the value record that binds to the variable name, and the bool reference is a mutable type used to track whether this variable has been used or not. The bool reference will initially be set to false. If there is any operation that refers or uses a quantum variable, the bool reference will be changed to true, indicating this quantum variable should not be used again.

This is an example of our approach.

```
let q = (new 0) in...
```

We declare a variable called q, and give it a binding value of 0 state. Then we will put the tuple of the variable name, 0 state, and false reference into our environment.

```
environment: [( ...     ...     ...     );
              ("q" * Qubit 0 * false ref);
              ( ...     ...     ...     )]
```

When later we use q for some operation, this binding in our environment will turn to q, 0 state, true reference.

```
environment: [( ...     ...     ...     );
              ("q" * Qubit 0 * true ref);
              ( ...     ...     ...     )]
```

We now present a violation of the no-cloning theorem and how *QAL* interpreter handles it.

```
1 let q1 = (new 0) in
2 let f x =
3   let q2 = (H q1) in
4   let g y = q2 in
5   g
6 in
7
8 (((f 0) 0), q1)
```

Listing 4: No-cloing Theorem Violation

In code fragment No-cloing Theorem Violation, we first declare a qubit variable q1 with state 0 on line 1. This will be the qubit that will eventually have a violation of the no-cloning theorem.

```
1 let q1 = (new 0) in
```

Then we define a function with an arbitrary parameter x. We declare a qubit variable q2 which will store the result of applying Hadamard gate on q1, now q1 should not be used anywhere in this program. We then return such an operation using a function called g and return the function g.

```
2 let f x =
3   let q2 = (H q1) in
4   let g y = q2 in
5   g
6 in
```

On line 8, we create a tuple with the first element of the tuple to call function f, which will eventually call function g that makes q1 unavailable to use again, and on the second element of the tuple, we want to use q1 again, here we violate the no-cloning theorem.

```
8 (((f 0) 0), q1)
```

When running No-cloing Theorem Violation code, we will see the output.

```
The quantum variable: 'q1' can not be used again!
```

Result of No-cloning Theorem Violation

The interpreter throws an error states that "The quantum variable: 'q1' can not be used again!" to indicate an error of the code that violates the no-cloning theorem.

3.5. Interpreter

The architecture of the interpreter to QAL is quite similar to the conventional ones shown in Figure 1.

Any QAL code files will first pass through a lexer to be converted into tokens, and then these tokens are parsed into an AST through the parser. The quantum data manager refers to the Quantum resource management, and here the core means the main body of the interpreter. When it is interpreting the AST, it calls functions from the standard library. All the important features such as closure, higher-order functions, and recursion are in the core of the interpreter.

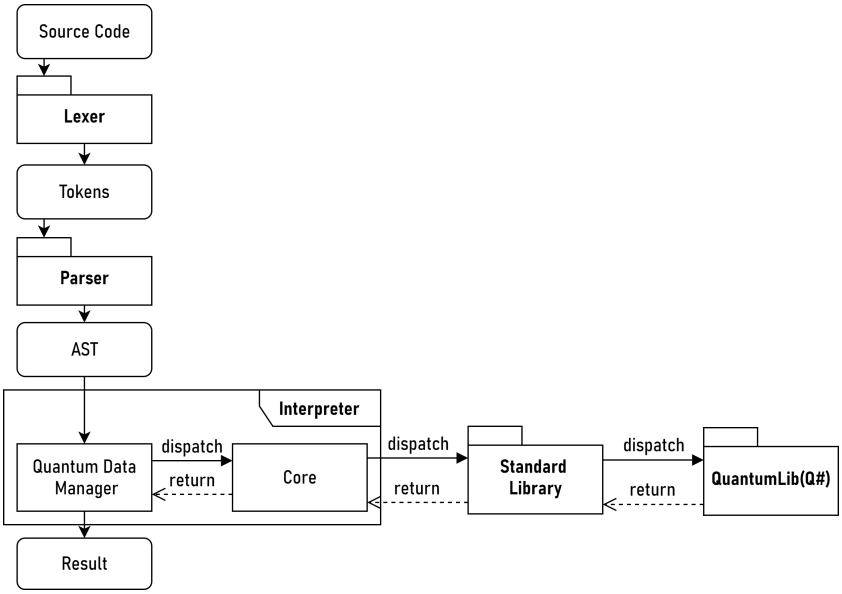

Figure 1: Flowchart of Interpreter

4. Future Developments

There are still a lot of features that are essential to a mature programming language while has yet to be implemented.

Currently, since we only focused on implementing the core features of *QAL*, the syntax and common utilities remained unpolished. As shown in the sample *QAL* code, the syntax is quite verbose. For example, there is no way of deconstructing a tuple using a let binding, and it is only possible to extract variables from a tuple by using an unnecessary match block. In addition, the aforementioned support for pattern matching recursive data structure should also be implemented in parser and interpreter.

Similarly, there are no infix operators, such as plus, minus, or multiply. Due to their absence, it is required to type in words a-d-d, m-i-n-u-s, etc., instead of their corresponding symbols. For now, it is for the sake of rapid development, but we should definitely make improvements in the future. Lastly, we need more syntactic sugar. For example, we can have the pipe and compose operator from F# to make the expressions more concise.

Moreover, we can implement the concept of entangled functions in QAL. The concept of entangled programs was first introduced in *Foundations of Quantum Programming*, that a quantum control block uses a set of orthogonal qubit states to form a superposition of the execution paths. As every "program" involved in functional programming languages are pure functions with no side effects, it is equivalent to entangled functions, which makes the exploration of this topic much easier. However, due to the limitation in our project schedule, we are not planning to implement this feature in the current version iteration, despite it being crucial to a high-level quantum programming language, as mentioned in many papers.

5. Conclusion

In general, we consider the progress made throughout this semester to be quite significant. We have successfully completed virtually every feature and specification within schedule, and the final product, the interpreter, is almost fully functional. Aside from proving the viability

of the concepts from the cutting-edge papers such as quantum recursion and quantum closure, we also implemented the idea of linearity, though not in its originally proposed way. In fact, all of these features worked perfectly with the fundamental concept of functional programming language.

We do recognize the imperfections that exist in the interpreter. Therefore, there is a great possibility that we would improve these in future versions. This is a link to our GitHub page where all commits are tracked: https://github.com/Quantumzhao/Quantum-Abstract-Language

References

[1] Dal Lago, U., Masini, A., and Zorzi, M. (2009). On a measurement-free quantum lambda calculus with classical control. *Math. Struct. Comput. Sci.*, 19(2):297–335.

[2] Dal Lago, U. and Rioli, A. (2015). Applicative bisimulation and quantum lambda-calculi (long version). *arXiv e-prints*, pages arXiv–1506.

[3] Gay, S. and Mackie, I. (2010). *Semantic techniques in quantum computation*. Cambridge University Press.

[4] Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., and Valiron, B. (2013). An introduction to quantum programming in quipper. In *International Conference on Reversible Computation*, pages 110–124. Springer.

[5] Saldyt, L. (2018). Qurry, a probabilistic quantum programming language.

[6] Selinger, P. and Valiron, B. (2009). Quantum lambda calculus. *Semantic Techniques in Quantum Computation*, page 135–172.

[7] Ying, M. (2016). *Foundations of Quantum Programming*. Elsevier.