Quantum Functional Programming

Cameron Moy, Andrew Witten, Suteerth Vishnu

December 13, 2018

1 Introduction

The λ-calculus is the theoretical foundation for functional programming. Researchers will often develop novel language features as extensions of the λ-calculus. Quantum programming is no exception. Selinger and Valiron developed the quantum λ-calculus, a linearly typed λ-calculus that incorporates quantum computation [6]. This calculus can ensure that incorrect quantum programs, for example those that clone qubits in violation of no-cloning, will be ill-typed. We will build up to this complexity in layers, by introducing

- the untyped λ-calculus,
- the simply-typed λ-calculus,
- the linear λ-calculus,
- and finally the quantum λ-calculus.

We will conclude with a discussion of some future areas of research in this space.

2 The untyped λ-calculus

The λ-calculus in its untyped form is a minimalistic programming language. Despite being a spartan language, it is Turing complete. Therefore, the Church-Turing thesis states that it is powerful enough to carry out any "real-world" algorithm. We describe the untyped λ-calculus in a similar presentation as [4].

A programming language definition is given in two parts: syntax and semantics. We define the syntax of a language as a context-free grammar, a formalism

$$M \to (\lambda X.M) \mid (M\ M) \mid X$$
$$V \to (\lambda X.M)$$

Figure 1: The syntax for the untyped λ-calculus.

that describes the set of abstract syntax trees making up programs in our language. The semantics of the language, at least in our presentation, is given as a small-step operational semantics describing how program pieces reduce, and as a grammar of evaluation contexts describing a deterministic reduction strategy. This technique, due to Felleisen and Hieb, subsumes lengthy congruence rules (like those found in [6]).

Figure 1 describes the syntax of the untyped λ-calculus. A λ term, denoted by the non-terminal M, is

- an abstraction $(\lambda X.M)$ which is interpreted as a function, or
- an application $(M\ M)$ which is interpreted as a function call, or
- a variable X which can range over some set of variable names.

The non-terminal V is just a subset of M that we will interpret as the values of our language.

Example. The λ abstraction $(\lambda x.x)$ represents the identity function. One can prove this term is in our language by way of a left-most derivation.

Proof.

$$\begin{aligned} M &\to (\lambda X.M) \\ &\to (\lambda x.M) \\ &\to (\lambda x.X) \\ &\to (\lambda x.x) \end{aligned}$$

❑

$$E \to \Box \mid (E\ M) \mid (V\ E)$$

Figure 2: The call-by-value grammar of evaluation contexts for the untyped λ-calculus.

$$(\beta)\frac{}{(\lambda x.m)\ v \to [x \mapsto v]m} \qquad (\Box)\frac{m \to m'}{e[m] \to e[m']}$$

Figure 3: The small-step operational semantics for the untyped λ-calculus.

To complete the description of the language we give an operational semantics. In figure 2 we specify a call-by-value deterministic evaluation strategy. This strategy evaluates application arguments eagerly, and does not reduce under λ abstraction bodies. This choice corresponds closely with the semantics of most everyday programming languages. In figure 3 we define the small-step operational semantics in natural deduction style. The β rule characterizes function application, and the $\Box$ rule characterizes sub-term reduction with the grammar of evaluation contexts. It is equivalent to all of the traditional call-by-value congruence rules.

Example. The λ term $(\lambda x.x\ x)\ (\lambda y.y)\ z$ reduces to z.

Proof.

$$\begin{aligned}(\lambda x.x\ x)\ (\lambda y.y)\ z &\to (\lambda y.y)\ (\lambda y.y)\ z \\ &\to (\lambda y.y)\ z \\ &\to z\end{aligned}$$

❑

$$
\begin{aligned}
M &\to (\lambda X : \boxed{T} \ . \ M) \mid (M \ M) \mid X \\
V &\to (\lambda X : \boxed{T} \ . \ M) \\
\boxed{T} &\to (T \to T) \\
\boxed{\Gamma} &\to \Gamma, X : T \mid \emptyset
\end{aligned}
$$

Figure 4: The syntax of the simply-typed λ-calculus.

3 The simply-typed λ-calculus

The untyped λ-calculus suffers from the problem that one can easily write non-sensical programs.

Example. The λ term $x \ (\lambda y.y)$ is a valid expression in our language, but does not evaluate to anything under our semantics, but is also not a value. We call this a stuck state.

The objective of our type system is to develop a syntactic method for identifying valid programs, in other words programs that will never reach a stuck state. We do this by

- formulating a typing relation on terms that will rule out all non-sensical programs, and
- proving a soundness theorem for this relation demonstrating that a well-typed program will never become stuck.

The syntax given in figure 4 extends the untyped λ-calculus syntax by adding type annotations. We highlight extensions in grey. We also have typing contexts Γ that are used in the typing relation given in figure 5. Modulo the type annotations, the semantics are exactly the same as the untyped λ-calculus.

Theorem (Soundness)**.** A well-typed term in the simply-typed λ-calculus will never reach a stuck state during evaluation.

$$\text{(App)}\frac{\Gamma \vdash m_1 : t_1 \to t_2 \quad \Gamma \vdash m_2 : t_1}{\Gamma \vdash (m_1\ m_2) : t_2} \qquad \text{(Abs)}\frac{\Gamma, x : t_1 \vdash m : t_2}{\Gamma \vdash (\lambda x : t_1\ .\ m) : t_1 \to t_2}$$

$$\text{(Var)}\frac{}{\Gamma_1, x : t, \Gamma_2 \vdash x : t}$$

Figure 5: The typing relation for the typed λ-calculus.

A well-typed term is one that is related to a type using the relation defined in figure 5. The proof of soundness is non-trivial and requires many additional theorems and lemmas, most notably the progress and preservation theorems. See [4] for details.

4 The linear λ-calculus

A linear type system ensures all linear objects are consumed exactly once. They have may uses, especially to track memory ownership. We present David Walker's formulation of the linear λ-calculus from [5]. In the simply-typed λ-calculus a number of structural properties are satisfied of typing judgments.

Lemma (Permutation)**.**

$$\frac{\Gamma_1, x_1 : t_1, x_2 : t_2, \Gamma_2 \vdash m : t}{\Gamma_1, x_2 : t_2, x_1 : t_1, \Gamma_2 \vdash m : t}$$

Lemma (Weakening)**.**

$$\frac{\Gamma_1, \Gamma_2 \vdash m : t}{\Gamma_1, x_1 : t_1, \Gamma_2 \vdash m : t}$$

Lemma (Contraction)**.**

$$\frac{\Gamma_1, x_1 : t_1, x_1 : t_1, \Gamma_2 \vdash m : t}{\Gamma_1, x_1 : t_1, \Gamma_2 \vdash m : t}$$

These structural properties can be proved by induction over typing derivations defined in figure 5. However, if we define the typing relation differently, we can construct a substructural type system, one that fails some of the above

$$
\begin{aligned}
M &\to Q\ (\lambda X : T\ .\ M) \mid (M\ M) \mid X \\
V &\to Q\ (\lambda X : T\ .\ M) \\
Q &\to \texttt{lin} \mid \texttt{un} \\
P &\to (T \to T) \\
T &\to Q\ P \\
\Gamma &\to \Gamma, X : T \mid \emptyset
\end{aligned}
$$

Figure 6: The syntax of the linear λ-calculus.

lemmas. A linear type system violates weakening and contraction. This has the consequence that every variable must be used exactly once, the desired property.

We need to define some new notions to make this work properly,

- a syntax that incorporates type qualifiers, specifying whether a value should be linear or unrestricted,
- a context-splitting relation that manages the typing context dependent on qualifiers,
- a subtyping relation between linear and unrestricted types used to define subsumption on types and contexts,
- and finally a new typing relation taking advantage of these relations.

We define the syntax of the linear λ-calculus in figure 6. This syntax allows us to annotate values and types with type qualifiers specified by Q. These qualifiers permit us to differentiate linear data `lin` and unrestricted data `un`.

We use the type qualifiers to define the context-splitting relation in figure 7. The context-splitting relation allows duplication of unrestricted types in rule `un`, but linear types must be partitioned in rules $\texttt{lin}_1$ and $\texttt{lin}_2$. This relation allows us to restrict typing contexts in the typing relation.

$$(\emptyset)\frac{}{\emptyset = \emptyset \circ \emptyset} \qquad (\texttt{un})\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \texttt{un}\ p = (\Gamma_1, x : \texttt{un}\ p) \circ (\Gamma_2, x : \texttt{un}\ p)}$$

$$(\texttt{lin}_1)\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \texttt{lin}\ p = (\Gamma_1, x : \texttt{lin}\ p) \circ \Gamma_2} \qquad (\texttt{lin}_2)\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \texttt{lin}\ p = \Gamma_1 \circ (\Gamma_2, x : \texttt{lin}\ p)}$$

Figure 7: The context-splitting relation for the linear λ-calculus.

$$(\text{Subtype})\frac{}{\texttt{lin} \leq \texttt{un}} \qquad (\text{Type})\frac{t = q'\ p \qquad q \leq q'}{q(t)} \qquad (\text{Context})\frac{(x : t) \in \Gamma}{q(\Gamma)}$$

Figure 8: The qualifier relation for the linear λ-calculus.

Finally, we formalize the relationship between `un` and `lin` types. This can be done with a subtyping relation and a type qualifier function over types and contexts. These are defined in figure 8.

These are all the tools necessary to define a new typing relation, one for the linear λ-calculus. This is done in figure 9. Highlighted are changes from the simply-typed λ-calculus.

$$(\text{App})\frac{\Gamma_1 \vdash m_1 : \texttt{q}\ t_1 \to t_2 \qquad \Gamma_2 \vdash m_2 : t_1}{\Gamma_1 \circ \Gamma_2 \vdash (m_1\ m_2) : t_2} \qquad (\text{Abs})\frac{\texttt{q}\ (\Gamma) \qquad \Gamma, x : t_1 \vdash m : t_2}{\Gamma \vdash \texttt{q}\ (\lambda x : t_1\ .\ m) : \texttt{q}\ t_1 \to t_2}$$

$$(\text{Var})\frac{\texttt{un}\ (\Gamma_1, \Gamma_2)}{\Gamma_1, x : t, \Gamma_2 \vdash x : t}$$

Figure 9: The typing relation for the linear λ-calculus.

$$
\begin{aligned}
B &\to 0 \mid 1 \\
M &\to Q\ (\lambda X : T\ .\ M) \mid (M\ M) \mid X \mid C \mid Q\ \langle M, M\rangle \mid Q\ B \mid Q\ \star \mid \ldots \\
V &\to Q\ (\lambda X : T\ .\ M) \mid C \mid Q\ \langle V, V\rangle \mid Q\ B \mid Q\ \star \mid \ldots \\
Q &\to \texttt{lin} \mid \texttt{un} \\
P &\to (T \to T) \mid T \otimes T \mid \texttt{Bit} \mid \texttt{Qubit} \mid \top \mid \ldots \\
T &\to Q\ P \\
\Gamma &\to \Gamma, X : T \mid \emptyset
\end{aligned}
$$

Figure 10: The syntax for some of the quantum λ-calculus.

5 The quantum λ-calculus

So far we have built up a sophisticated type system, but unfortunately it is not useful. The quantum λ-calculus will augment our calculus with primitives, both classical and quantum.

The quantum λ-calculus uses a linear type system to guarantee the no cloning theorem. In a quantum program, one shouldn't be able to a copy of a quantum state [1]. A linear type system can enforce this restriction.

The no cloning theorem states that it is impossible to duplicate non-orthogonal quantum states. In other words, no unitary transformation U can act on non-orthogonal states $|\psi\rangle$ and $|\phi\rangle$, such that $U(|\psi\rangle|\phi\rangle) = |\psi\rangle|\psi\rangle$ or $U(|\psi\rangle|\phi\rangle) = |\phi\rangle|\phi\rangle$. It is possible to duplicate orthogonal states [2]. The type system for any quantum programming language should not allow a quantum state to be duplicated. For example, the function $\lambda x : t\ .\ \langle x, x\rangle$, where $\langle \cdot, \cdot \rangle$ is a pair, only makes sense when x can be copied (in other words, it's classical not quantum).

Our development of the quantum λ-calculus follows that of [6], although presented in the style of [5]. To give interesting examples we will add to the syntax of the simply typed λ-calculus some ML-like forms. This syntax is developed in

$$\text{(App)}\ \frac{\Gamma_1 \vdash m_1 : \mathtt{q}\ t_1 \to t_2 \qquad \Gamma_2 \vdash m_2 : t_1}{\Gamma_1 \circ \Gamma_2 \vdash (m_1\ m_2) : t_2} \qquad \text{(Abs)}\ \frac{\mathtt{q}(\Gamma) \qquad \Gamma, x : t_1 \vdash m : t_2}{\Gamma \vdash \mathtt{q}\ \ (\lambda x : t_1\ .\ m) : \mathtt{q}\ \ t_1 \to t_2}$$

$$\text{(Var)}\ \frac{\mathtt{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x : t, \Gamma_2 \vdash x : t} \qquad \text{(Pair)}\ \frac{\Gamma_1 \vdash m_1 : t_1 \qquad \Gamma_2 \vdash m_2 : t_2 \qquad \mathtt{q}(t_1) \qquad \mathtt{q}(t_2)}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{q}\ \ \langle m_1, m_2 \rangle : \mathtt{q}\ \ t_1 \otimes t_2}$$

$$\text{(Bit)}\ \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathtt{q}\ \ b : \mathtt{q}\ \ \mathtt{Bit}} \qquad \text{(Unit)}\ \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathtt{q}\ \star : \mathtt{q}\ \top}$$

Figure 11: The typing relation for some of the quantum λ-calculus.

figure 10. The added forms are C a set of constants, $\langle M, M \rangle$ pairs, booleans, and the unit. For brevity, we elide some other helpful forms and leave the `lin` qualifier implicit.

Now, we type the new forms in figure 11. With this typing relation we can show some examples of terms that are well-typed and ill-typed.

Example. The term $\lambda x : \top\ .\ (meas\ (hada\ (new\ 0)))$ has type $\top \to \mathtt{Bit}$.

Proof.

$$\text{(Abs)}\ \frac{\text{(App)}\ \dfrac{\text{(Meas)}\ \dfrac{}{x : \top \vdash meas : \mathtt{Qubit} \to \mathtt{Bit}} \qquad \text{(App)}\ \dfrac{\dfrac{}{\emptyset \vdash hada : \mathtt{Qubit} \to \mathtt{Qubit}} \qquad \text{(App)}\ \dfrac{\dfrac{}{\emptyset \vdash new : \mathtt{Bit} \to \mathtt{Qubit}} \qquad \dfrac{}{\emptyset \vdash 0 : \mathtt{Bit}}}{\emptyset \vdash new\ 0 : \mathtt{Qubit}}}{\emptyset \vdash hada\ (new\ 0) : \mathtt{Qubit}}}{x : \top \vdash (meas\ (hada\ (new\ 0))) : \mathtt{Bit}}}{\emptyset \vdash \lambda x : \top\ .\ (meas\ (hada\ (new\ 0))) : \top \to \mathtt{Bit}}$$

❑

Let's show that an attempt to clone a qubit fails. The λ term $\lambda x : \mathtt{Qubit}\ .\ \langle x, x \rangle$ clones a qubit by constructing a pair. Therefore, this term cannot possibly have type $\mathtt{Qubit} \to \mathtt{Qubit} \otimes \mathtt{Qubit}$. An attempt to prove this typing judgement shows why this cannot possibly hold.

Proposition. $\lambda x . \langle x, x \rangle \not: \mathtt{Qubit} \to \mathtt{Qubit} \otimes \mathtt{Qubit}$

Solution.

$$\dfrac{\dfrac{\dfrac{}{x : \texttt{Qubit} \vdash x : \texttt{Qubit}}\,(\text{Ok}) \quad \dfrac{}{\emptyset \vdash x \not: \texttt{Qubit}}\,(\text{Bad})}{x : \texttt{Qubit} \vdash \langle x, x \rangle : \texttt{Qubit} \otimes \texttt{Qubit}}\,(\text{Pair})}{\emptyset \vdash \lambda x.\langle x, x \rangle : \texttt{Qubit} \to \texttt{Qubit} \otimes \texttt{Qubit}}\,(\text{Abs})$$

❑

When we apply the pair rule we must partition the typing context Γ. We must choose either the left or right component of the pair to populate with the element in the context. In our example above we choose the left side. However, this leaves the other side with an empty typing context. One can see this from our "bad" inference. This makes it ill-typed.

6 Recent and future work

Recently, Paykin published a dissertation that discusses an interface that associates, for every non-linear type α, a linear type **Linear** α, and for every linear type σ, a non-linear type **Lift** σ. **Lower** and **Lift** can be viewed as operators that take a non-linear type to a linear type, and a linear type to a non-linear type, respectively. In this linear/non-linear (LNL) type system, linear and non-linear data are on equal ground. This interface has implications for quantum programming. Paykin discusses the denotational semantics of higher-order quantum functions and, in particular, describes how to map quantum programs to superoperators over density matrices [3].

For future work, there is an open problem is formally showing that the quantum λ-calculus is equivalent to a universal quantum Turing machine. It is conjectured that this is true, but a complete proof has yet to be provided [7]. Despite the high-level presentation above, one must eventually compile the language down into a lower-level representation. Compiler optimizations may be necessary. In spite of the extensive research done on quantum circuits, the analysis for quantum program optimizations for high-level languages is sparse [8]. Finally, control flow in the presented λ-calculus, and in the literature, is classical. An unexplored area is considering functional quantum control [8].

References

[1] P. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[2] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[3] J. Paykin. *Linear/Non-Linear Types for Embedded Domain-Specific Languages*. PhD thesis, University of Pennsylvania, 2018.

[4] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[5] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[6] P. Selinger, B. Valiron, et al. Quantum lambda calculus. *Semantic Techniques in Quantum Computation*, pages 135–172, 2010.

[7] A. Van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004.

[8] M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.