Linear Dependent Type Theory for Quantum Computation

Arnav Dandu
adandu@ucsd.edu

Ryan Batubara
rbatubara@ucsd.edu

CSE 291: Quantum Complexity - University of California, San Diego

Abstract

A type theory gives rise to a threefold perspective on computer programs: As a program or algebra, as a type, and as categories. By enforcing a type system onto quantum programs, error checking can be done more easily at compile time, avoiding the probabilistic nature of quantum measurement and run-time verification. However, classical homotopy type theory fails to account for quantum mechanics such as no-cloning. We explore linear dependent type theory, how it deals with quantum mechanics such as no-cloning, and its implications to quantum complexity. We then discuss alternatives and extensions to linear dependent type theory for the foundation of quantum programming languages.

1 Introduction

We begin by giving relevant definitions in type theory, with the intention of building up relatively self-contained rigor for proofs in Section 2, where we prove the relationship between various type theories and quantum mechanics, particularly the no-cloning theorem. Section 3 then reviews alternatives to linear dependent type theory for the basis of quantum computation.

1.1 Introducing Type Theory (TT)

Type Theory (TT) presents an alternative foundation to mathematics compared to Set Theory (ST). The key difference is that TT presents syntactic information while ST presents it semantically [1] [22]. We attribute all the following definitions to [22]. In TT, a value a of type A is denoted $a : A$. Define $\mathbb{U}_0$ as a universe of types, such that $A : \mathbb{U}_0$. In order to prevent Russell's paradox as with sets, we define an infinite series of type universes such that

$$\mathbb{U}_0 : \mathbb{U}_1 : \mathbb{U}_2 : \mathbb{U}_3 : \cdots \tag{1}$$

and say that a type $A : \mathbb{U}$ if we have $A : \mathbb{U}_i$ for some $i \in \mathbb{N} \cup \{0\}$. Mappings between types, such as $A \to B$ for types A, B are types as well. Implications are denoted using $\vdash$, such as

$$x : \mathbb{N} \vdash (x + 1) : \mathbb{N} \tag{2}$$

which says if x is of type natural number, then $x + 1$ is as well. In TT, $\mathbb{N}$ is defined similarly to in ST. Namely, define the base $0 : \mathbb{N}$ and a successor function $\text{succ} : \mathbb{N} \to \mathbb{N}$. Therefore all natural numbers besides 0 can be defined as

$$1 \equiv \text{succ}(0) \tag{3}$$

$$2 \equiv \text{succ}(\text{succ}(0)) \tag{4}$$

$$\dots \tag{5}$$

We can also create tuples of types (A, B) such that $(a, b) : (A, B)$. Indeed, other structures for combining types such as

$$A \otimes B,\ \sum A,\ \prod A,\ A \times B,\ \dots \tag{6}$$

can be defined and much structure can be gained this way. We say these augment or extend the type theory. However, as we will see we need to be careful about what we augment the basic TT with, since sometimes excluding these structures (or redefining them) can lead to a TT better suited for our purposes, namely computing.

[1] We use Church's TT, not Curry's. See [22].

1.2 Classical Motivation for TT

Type theory was largely developed to aid the verification or proof of computer programs [4]. An informal example of this, for instance, would be a compiler of a strongly typed language that checks if the outputs of a function are all the same type, and errors otherwise. More formally, one can denote logical propositions as types.

We first begin by specifying a key difference between saying $s \in S$ in ST and $a : A$ in TT. The complete proof and list of properties is out of scope (see [11]), but the key property we need is that $\in$ can be undecidable, but a is decidable [2] [11]. This is because to show $s \in S$ requires a proof, but $a : A$ is a proof in itself in the following sense:

Let A denote the proposition "A Quantum Computer exists." Then a statement a witnessing "UCSD has a quantum computer" proves A. More generally, base type theory has a correspondence to "intuitionistic" logic: types A are propositions, $a : A$ are proofs, and $A \to B$ corresponds to $A \Rightarrow B$. For more correspondences, see 1.5.

1.3 Dependent Type Theory (DTT)

Dependent Type Theory (DTT) augments TT with types that can depend on values of other types, rather than just the types themselves [22]. This removes the distinction between values and types in TT. For example, given

$$A \xrightarrow{B} \mathbb{U} \tag{7}$$

define values of a dependent pair type

$$(a, b) : \sum_{x:A} B(x) \tag{8}$$

where $a : A$ and $b : B(x)$. We can think of B as a map from values of A to values[3] of other types in $\mathbb{U}$. We can correspondingly define $\mathrm{Pr}_1 : \sum_{x:A} B(x) \to A$ that maps values of types $B(x) : \mathbb{U}$ to values $a : A$[4].

[2] There are type systems with undecidable :, such as the Curry variant of system F, but we do not focus on such systems here. See [11].

[3] Note these values can be types themselves due to DTT.

[4] Pr_1 is the standard notation.

Suppose we know that $(a, b) : \sum_{x:A} B(x)$. Therefore, we have a value $a : A$ with corresonding $B(a) : \mathbb{U}$. In terms of logic, this means that there exists a value a such that $B(a)$ also is defined. Continuing the example in 1.2, $B(x)$ can map from universities to the quantum computers (QC) they have, so

$$\sum_{x:A} B(x) \equiv \text{"A univ. has QC"} \tag{9}$$

$$(a, b) \equiv (\text{"UCSD"}, \text{"UCSD has QC}) \tag{10}$$

would prove $\sum_{x:A} B(x)$. That is, $\sum_{x:A} B(x)$ corresponds to the proposition $\exists(x : A).B(x)$ [5]. Indeed, if $B(x)$ is constant we have

$$\sum_{x:A} B \equiv A \times B \tag{11}$$

Furthermore, $\prod_{x:A} B(x)$ corresponds to $\forall(x : A).B(x)$. In other words, providing a value for such a type proves $B(x)$ for all $x : A$. Continuning the previous example

$$((\text{"UCSD"}, \text{"UT"}), \tag{12}$$
$$(\text{"UCSD has QC"}, \text{"UT has QC"})) : \prod_{x:A} B(x)$$

assuming A only has these two values proves the proposition "All universities in A have a QC." Indeed, if $B(x)$ is constant we have

$$\prod_{x:A} B \equiv A \to B \tag{13}$$

because each $a : A$ corresponds to a $b : B$.

1.4 Homotopy Type Theory (HoTT)

Homotopy Type Theory (HoTT) further augments DTT with a topological perspective, which is particularly important when looking its homotopical definition of identity [22]. We begin by defining DTT in its most rigorous form. A context Γ is a list of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$$

[5] We use the lambda calculus notation here, instead of the usual set notation, to further distinguish TT and ST.

defines variables of different types. Let . be the empty context. This gives rise to judgments, which derive statements as seen in Table 1

Judgment	Meaning
Γ ctx	Γ is a well-formed context.
$\Gamma \vdash a : A$	Given Γ, term a has type A.
$\Gamma \vdash a \equiv a' : A$	Given Γ, a, a' equal by def

Table 1: (Some) Judgments in HoTT

Here, a well-formed context means that it is proven. Namely, no type in a well-formed context can be empty. Given these judgments, we use inference rules to derive further judgments using the notation

$$\frac{\Gamma_1, \ldots, \Gamma_n}{\Gamma} \text{ Inference Name}$$

There are many inference rules[6], but we summarize the ones necessary for us below. First we have the initialization of an empty context:

$$\frac{}{. \text{ ctx}} \text{ ctx EMP}$$

We also have the initialization of type universes:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbb{U}_i : \mathbb{U}_{i+1}} \mathbb{U} \text{ Intro}$$

$$\frac{\Gamma \vdash A : \mathbb{U}_i}{\Gamma \vdash A : \mathbb{U}_{i+1}} \mathbb{U} \text{ Cummul}$$

We also have the ability to introduce a value of a new type, given that type is a value of some type universe:

$$\frac{x_1 : A_1, \ldots, x_{n-1} : A_{n-1} \vdash A_n : \mathbb{U}}{x_1 : A_1, \ldots, x_{n-1} : A_{n-1}, x_n : A_n \text{ ctx}} \text{ ctx EXT}$$

We then have the ability to initialize variables from types

$$\frac{x_1 : A_1, \ldots, x_n : A_n \text{ ctx}}{x_i : A_i \text{ ctx}} \text{ Vble } (1 \leq i \leq n)$$

We can also weaken, which replaces $A : \mathbb{U}$ with $a : A$, since a is a "proof" of A:

$$\frac{\Gamma \vdash A : \mathbb{U} \qquad \Gamma, \Delta \vdash b : B}{\Gamma, x : A, \Delta \vdash b : B} \text{ wkg}$$

[6]There are many more inference rules than there are modus ponens in logic, but they are similar.

We can also substitute. For instance, given a statement using $x : A$ in the RHS, replace all instances of x with a as denoted by $[a/x]$:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]} \text{ Subst}$$

This gives rise to a formal definition of $\equiv$.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \text{ def}$$

We have $\equiv$ have the following inferences:

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash a : A} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A}$$

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B : \mathbb{U}}{\Gamma \vdash a : B}$$

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash A \equiv B : \mathbb{U}}{\Gamma \vdash a \equiv b : B}$$

These allow us to formally introduce the topological value of HoTT[7]. Consider types as topological spaces, values as points, and maps between values as paths between values in the space. Then

$$\frac{\Gamma \vdash A : \mathbb{U} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv_A b : \mathbb{U}}$$

says that a and b are homotopically equivalent under A, denoted $a \equiv_A b$, if there exists a path $a \to b$ and $b \to a$ in A. To define this more conretely, we need a homotopical definition of idenity, given as refl defined

$$\frac{\Gamma \vdash A : \mathbb{U} \qquad \Gamma \vdash a : A}{\Gamma \vdash \text{refl}_a : a \equiv_A a}$$

Visually, refl_a is a the path $a \to a$ in A:

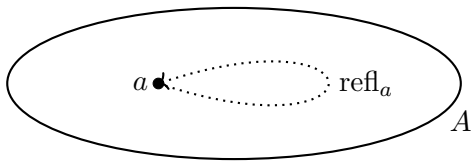


Figure 1: refl_a

[7]Note that this enforces a necessary topological structure on types, something ST does not enforce with sets. See FREEDMAN or WANG.

Then for a type family of paths $P : A \to \mathbb{U}$ and $f, g : \prod_{x:A} P(x)$, define a homotopy from f to g denoted $f \sim g$ as a dependent function of type

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x)) \tag{14}$$

Note that this is defined up to equivalences. As we will see, these paths can be different yet equivalent up to homotopy.

Given $f : A \to B$, we then have a "type of proof" to show the equivalence of f in the type[8]

$$\text{isequiv(f)} :\equiv \tag{15}$$
$$(\textstyle\sum_{g:B\to A}(f \circ g \sim \mathrm{id}_B)) \times (\sum_{h:B\to A}(h \circ f \sim \mathrm{id}_A))$$

where $\circ$ is the usual composition. This is much clearer visually, where A is a disc and A' is a torus topologically:

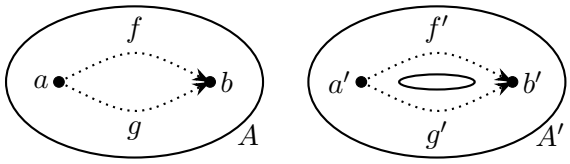


Figure 2: $f \sim g$ and $f' \not\sim g'$

This gives us a new alternative definition for the equivalence of types:

$$(A \simeq B) :\equiv \sum_{f:A\to B} \text{isequiv}(f) \tag{16}$$

such that two types are equivalent if all their paths are equivalent. Thus in Figure 2 $A \not\simeq A'$.

This finally allows us to deifne the fundamental axioms of HoTT (which we will show contradict if we attempt to enforce no-cloning in Section 2). First, let $f, g : \prod_{x:A} B(x)$ with $B : A \to \mathbb{U}$. Then there exists

$$\text{happly} : (f = g) \to \prod_{x:A} (f(x) =_{B(x)} g(x)) \tag{17}$$

which basically states that two paths are equal if they are pointwise equal in the topological perspective. The Fundamental Existentionality Axiom (FEA) states that there exists

$$\text{funext} : \prod_{x:A} (f(x) =_{B(x)} g(x)) \to (f = g) \tag{18}$$

[8]Here we use id instead of refl. These are different but we can gloss over this.

along with homotopies α, β such that

$$(\text{funext}, \alpha, \text{funext}, \beta) : \text{isequiv(happly)}. \tag{19}$$

This formally defines a correspondence between equal paths as paths that are pointwise equal. So in Figure 2, though $f \sim g$, $f \neq g$. Furthermore, the Univalence Axiom (UA) states that there exists

$$\text{idhequiv} : (A =_{\mathbb{U}} B) \to (A \simeq B) \tag{20}$$

which is a correspondence between equals and equivalence, such that there exists a map of the opposite direction

$$\text{ua} : (A \simeq B) \to (A =_{\mathbb{U}} B) \tag{21}$$

along with homotopies α', β' such that

$$(\text{ua}, \alpha', \text{ua}, \beta') : \text{isequiv(idhequiv)}. \tag{22}$$

The UA essentially says that $(A =_{\mathbb{U}} B) \simeq (A \simeq B)$. One way of thikning about the UA is to see it as a definition of isomorphisms in HoTT.

1.5 Computational Trinitarianism

In the previous sections, we have attemped to provide a rigorous buildup to HoTT. At the same time, we see how some concepts in HoTT are much more easily presented (and proven) when viewed from the lens of category theory or of topology. Briefly, a category C consists of a set of objects $X \in C$ such that for any $X, Y \in C$, then we have a set of morphisms (maps) dentoted $\mathrm{Hom}(X, Y)$ satisfying:

1. For all objects, there exists the identity morphism $1_X \in \mathrm{Hom}(X, X)$.
2. For $f : X \to Y$ and $g : Y \to Z$, there exists composite morphism $gf : X \to Z$.
3. Given morphisms h, g, f, $(hg)f = h(gf)$.

Putting it all together, we come to the concept of Computational Trinitarianism, summarized in Figure 3. Essentially, Computational Trinitarianism states that Category Theory, Type Theory, and Logic / Algebra are all equivalent and interchangeable perspectives on computation [22].

This means that a proof in one is a proof in the other two; we can choose what to use in proofs without loss of generality. We will make extensive use of this in Section 2.

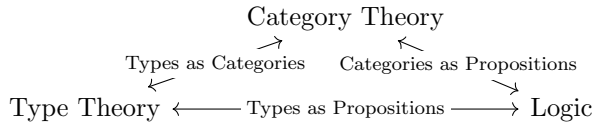


Figure 3: Computational Trinitarianism

One can then create more specific trinitarianisms by further specifying one of the three theories. For example:

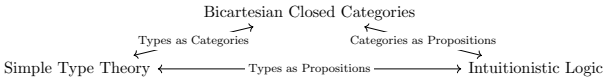


Figure 4: Simple TT Trinitarianism

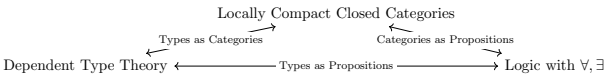


Figure 5: Dependent TT Trinitarianism

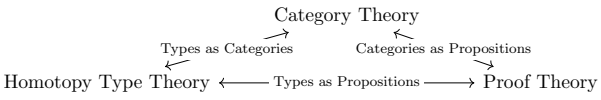


Figure 6: Homotopy TT Trinitarianism

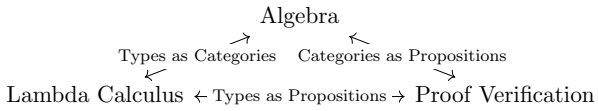


Figure 7: Rephrased HoTT Trinitarianism

Recall that our goal is to present a type theory that is well suited for Quantum Programs. By computational trinitarianism, it sufficies to show that in one of the above theories, enforcing quantum mechanics leads to problems to show that HoTT is not an appropriate type theory for Quantum. Similarly, if we can show one some other TT must be consistent with one of the theories, then there must be a way to make it consistent with all the theories!

2 A Quantum Type Theory

We seek a type theory that faithfully captures the principles of quantum logic. A substructural logic known as linear logic is considered the basis of a quantum logic due its omission of the contraction rule and weakening rule, reflecting no-cloning and no-deleting [10]. Linear type theory is then the linear logic-version of type theory. We begin by demonstrating why HoTT is insufficient as a framework for quantum logic.

2.1 No-Deleting in HoTT

We offer two proofs for the incompatibility of HoTT with quantum computing. The first comes from contraction and no-deleting [10].

Proof. Type A is contractible if there is $a : A$ called the center of the contraction such that $a = x$ for all $x : A$. More formally,

$$\text{isContr}(A) :\equiv \sum_{a:A} \prod_{x:A} (a = x) \tag{23}$$

Contractibility is a necessary property of HoTT since refl exists and therefore an identity must exist. More concretely, a contractible type $A \to A$ with a single element is the trivial solution to both FEA and UA, and therefore must exist as long as both axioms are enforced.

Lemma: If $P : A \to \mathbb{U}$ is a type family such that each $P(a)$ is contractible, then

$$\prod_{x:A} P(x) \tag{24}$$

is contractible[9].

Proof. $\prod_{x:A} P(x)$ is a proposition, and it has a single element of the form of a map that sends every $x : A$ to the center of contraction of $P(x)$. Therefore, this element witnesses that $\prod_{x:A} P(x)$ is contractible. □

Lemma: For any A and any $a : A$,

$$\sum_{x:A} (a = x) \tag{25}$$

is contractible[10].

[9] Lemma 3.11.6 in [22].
[10] Lemma 3.11.8 in [22].

Proof. Choose the center to be (a, refl_a). Suppose $(x, p) : \sum_{x:A}(a = x)$. We show $(x, p) = (a, \text{refl}_a)$. It suffices to show $q : a = x$ such that $q(\text{refl}_a) = p$, that is they are equivalent. But just take $q :\equiv p$ (which must exist since p). □

Putting it all together, we have that any type that has at least a single value, we can construct a contractible type out of it (by the second Lemma). Consider the type qubit, which has more than one value. But then it is possible to contract this type to one that has only a single value, losing information in the process. □

2.2 No-Cloning in HoTT

We claim no-cloning and Hott are incompatible as seen in [10].

Proof. Any quantum system that uses qubits must have a type to represent it. If the type qubit exists, then it must have a value. But then one can repeatedly apply $\mathbb{U}$ Intro, ctxETX, and Vble to initialize multiple copies of the same qubit. We can verify that these are the same qubit via $\equiv$. But then we can copy quantum information, since any initialized qubit must have a type. □

2.3 Linear Type Theory (LTT)

We use the definition outlined in [23]. In our context Γ, we have

$$\Gamma := x_1 : T_1, \ldots, x_n : T_n. \tag{26}$$

Our type system possesses *unicity of type*, which means that for a given Γ and t there is at most T satisfying $\Gamma \vdash t : T$. Still, in conventional type theories corresponding to intuitionistic logic, the existence of the contraction rule allows us to use a variable $x : T$ arbitrarily many times. Thus,

$$x : T \vdash () : \top \tag{27}$$

is valid though we never use the assumption, and

$$x : T \vdash (x, x) : T \times T \tag{28}$$

is valid even if we use the assumption $x : T$ twice.

A linear type system asserts that each assumption is used exactly once, making the two typings described above illegal. This means we must modify the inference rules from dependent type theory. For example, the variable rule

$$\frac{}{\Gamma, x : T \vdash x : T}\ \text{Var}$$

is illegal in LTT as we don't use any assumptions from Γ in the typing. We fix this by changing the rule to

$$\frac{}{x : T \vdash x : T}\ \text{Var}$$

so that the assumption $x : T$ is used exactly once. For brevity, we won't go over the analogues of the other inference rules, but the main point is that linear type theory prevents duplication and deletion of resources.

Although linear type theory and dependent type theory each have well-defined syntaxes individually, the precise syntax of a combined linear dependent type theory is still under development. However, since the categorical semantics for such a type theory are already established, we will not concern ourselves with this gap.

2.4 Categorial Semantics of a Linear Dependent Type Theory (LDTT)

Linear dependent type theory is precisely the formal language for which symmetric closed monoidal categories are the semantics [15]. A rigorous justification for this correspondence is out of scope, but an intuitive explanation can be proved.

A symmetric monoidal category is a category $\mathcal{C}$ equipped with a bifunctor

$$\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C} \tag{29}$$

called the tensor product, and an object I, called the unit object. There are natural isomorphisms expressing associativity

$$a_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C), \tag{30}$$

left and right unit laws

$$l_A : I \otimes A \simeq A \quad r_A : A \otimes I \simeq A, \tag{31}$$

and symmetry

$$\sigma_{A,A} : A \otimes B \simeq B \otimes A. \tag{32}$$

These isomorphisms must satisfy coherence conditions (namely, any diagram containing only a, l, r must commute) that ensure all ways of regrouping and reordering produce canonically isomorphic results.

A symmetric monoidal category is "closed" if, for each object A, the functor $- \otimes A$ has a right adjoint $[A, -]$. This structure provides a canonical way to talk about "linear functions" as objects themselves, mirroring the linear function types in the theory.

This alignment means that just as regular dependent type theory is modeled by categories that freely allow duplication and disposal of data (cartesian closed categories), linear dependent type theory is modeled by categories that enforce strict accounting of resources (symmetric monoidal closed categories).

2.5 No-Cloning in LDTT

We prove that no-cloning holds in LDTT by proving that it holds in a symmetric monoidal category, following the proof of Theorem 11 in [1] with some simplifications.

We begin by providing an axiomatic description of cloning in this setting.

Let $\mathcal{C}$ and $\mathcal{D}$ be monoidal categories. A monoidal functor $(F, e, m) : \mathcal{C} \to \mathcal{D}$ consists of a functor $F : \mathcal{C} \to \mathcal{D}$, an isomorphism $e : I \simeq FI$, and a natural isomorphism $m_{A,B} : FA \otimes FB \to F(A \otimes B)$.

For monoidal functors $(F, e, m), (G, e', m') : \mathcal{C} \to \mathcal{D}$, a monoidal natural transformation is natural transformation $t : F \to G$ such that the following diagrams commute:

$$\begin{array}{ccc} I & \xrightarrow{e} & FI \\ & {\scriptstyle e'}\searrow & \downarrow{\scriptstyle t_I} \\ & & GI \end{array}$$

$$\begin{array}{ccc} FA \otimes FB & \xrightarrow{m_{A,B}} & F(A \otimes B) \\ {\scriptstyle t_A \otimes t_B}\downarrow & & \downarrow{\scriptstyle t_{A\otimes B}} \\ GA \otimes GB & \xrightarrow[m'_{A,B}]{} & G(A \otimes B) \end{array}$$

We then say that a monoidal category has uniform cloning if it has a diagonal, which is a monoidal transformation

$$\Delta_A : A \to A \otimes A$$

such that Δ_A is coassociative

$$\begin{array}{ccccc} A & \xrightarrow{\Delta} & A \otimes A & \xrightarrow{1\otimes\Delta} & A \otimes (A \otimes A) \\ \| & & & & \downarrow{\scriptstyle a_{A,A,A}} \\ A & \xrightarrow{\Delta} & A \otimes A & \xrightarrow{\Delta\otimes 1} & (A \otimes A) \otimes A \end{array}$$

and cocommutative

$$\begin{array}{ccc} A & \xrightarrow{\Delta} & A \otimes A \\ & {\scriptstyle \Delta}\searrow & \downarrow{\scriptstyle \sigma_{A,A}} \\ & & A \otimes A \end{array}$$

We will now prove the following:

Theorem 1. *Let $\mathcal{C}$ be a compact closed category with cloning. Then every endomorphism is a scalar multiple of the identity; that is, for any $f : A \to A$, $f = \mathsf{Tr}(f) \cdot a$.*

A compact closed category is a symmetric monoidal category in which every object A has a dual A^*, and a unit and counit

$$\eta_A : I \to A^* \otimes A \quad \epsilon_A : A \otimes A^* \to I \tag{33}$$

such that diagram

$$\begin{array}{ccccc} A & \xrightarrow{r_A^{-1}} & A \otimes I & \xrightarrow{1_A\otimes\eta_A} & A \otimes (A^* \otimes A) \\ {\scriptstyle 1_A}\downarrow & & & & \downarrow{\scriptstyle a_{A,A^*,A}} \\ A & \xleftarrow[l_A]{} & I \otimes A & \xleftarrow[\epsilon_A\otimes 1_A]{} & (A \otimes A^*) \otimes A \end{array}$$

and the dual for A^* commute. It is clear that

$$(\epsilon_A \otimes 1_A) \circ (1_A \otimes \eta_A) = 1_A \tag{34}$$

and

$$(1_{A^*} \otimes \epsilon_A) \circ (\eta_A \otimes 1_{A^*}) = 1_A. \tag{35}$$

We start with the following lemma.

Lemma 2. *Let $u : I \to A \otimes B$ be a morphism in a symmetric monoidal category with cloning. Then*

$$u \otimes u = (3214) \otimes (u \otimes u).$$

The proof of this fact follows from an uninteresting diagram chase.

Symbolically, this means that we can say

$$u \otimes u : I \otimes I \to (A \otimes B) \otimes (A \otimes B) \tag{36}$$

is equivalent to

$$\sigma \circ (u \otimes u) : I \otimes I \to A \otimes (A \otimes B) \otimes B \tag{37}$$

This then gives us that for $\eta_A : I \to A^* \otimes A$,

$$\eta_A \otimes \eta_A : I \otimes I \to (A^* \otimes A) \otimes (A^* \otimes A) \tag{38}$$

is equal to

$$\sigma \circ (\eta_A \otimes \eta_A) : I \otimes I \to A^* \otimes (A^* \otimes A) \otimes A \tag{39}$$

We now have all we need to prove Theorem 1.

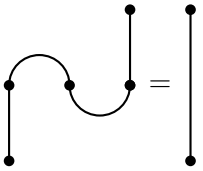
Proof. We prove this diagrammatically using Penrose graphical notation. We write units as

$$\eta_A : I \to A^* \otimes A$$

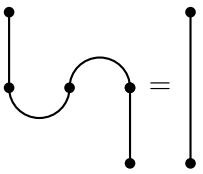
and counits as

$$\epsilon_A : A \otimes A^* \to I$$

and the identities (34) and (35) as

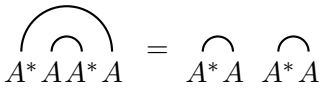


and

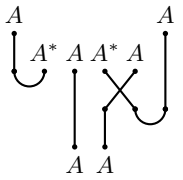


Think of this as pulling the string diagram taut. In general, when you can do this, there is an equivalence.

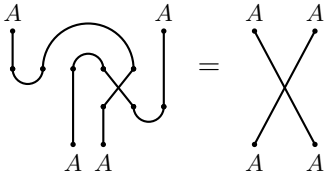
Our result from lemma 2 has the diagram



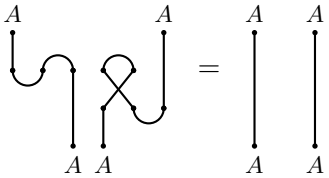
Now, consider the context



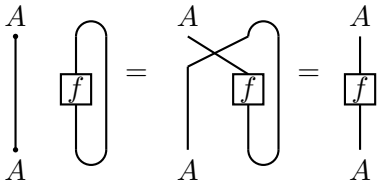
We then obtain



and



Then we have for any endomorphism $f : A \to A$,



which implies $f = \mathsf{Tr}(f) \cdot 1_A$. □

Applying this result for the natural setting of quantum computation, Vect_k, gives us no-cloning as it implies all linear maps within a vector space are just scalar multiples of the identity, preventing any non-trivial quantum operations.

3 QT for Quantum Computing

We have established that LDTT is a viable TT for quantum computing. Here, we take a higher level sketch of how LDTT may be useful for quantum computing.

3.1 Quantum Trinitarianism (QT)

Perhaps the most tangible benefit of a type theory for computing is using it as a framework for proof and program verification. In other words, we need to specify what categories and logic correspond to LDTT.

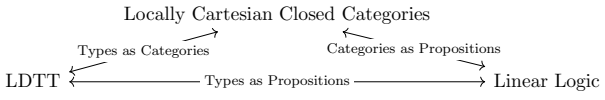


Figure 8: Quantum Trinitarianism (QT)

We have already seen that symmetric monoidal categories, or more precisely fibrations of symmetric monoidal categories or equivalently locally cartesian closed categories (LCCC), correspond to LDTT nicely. What takes the place of logic is Linear Logic, which though we do not define here, essentially represent the logic governing quantum systems. Note that circuit-description-based quantum computing is a contender for replacing linear logic, but pure quantum circuits lack a notion of dependent types.

3.2 LDTT for Classical-Quantum

We take this opportunity to finally handle a very important detail: Quantum systems are typically classically controlled, where classical information is not subject to no-cloning and other quantum restrictions. This means that though LDTT is a viable basis for the analysis, verification, and construction of purely quantum algorithms, it has limited practical benefit as it is too restrictive for classical types.

However, what we learn from LDTT is quantum trinitarianism, namely the interchangeability of perspectives on a quantum program. Embedding information from QT into HoTT based, classical systems therefore becomes a viable method of analyzing, verifying, and constructing quantum programs with classical control[11]. For a description of such a system, see the Appendix or [16]. Hence, we will shift into reviewing literature that use some form of trinitarianism (or parts of it) for quantum computing.

[11] Unfortunately, this is far beyond the scope of the paper!

3.3 QT for Quantum Programming

We showed earlier that (traditional) Lambda Calculus corresponds to HoTT in trinitarianism, which by extension makes it unsuitable as a computational model for quantum computation. However, with some modifications, a lambda calculus can be made for quantum programs: In [19], Selinger and Valiron use TT to extend lambda calculus with quantum types and operations. A key highlight of their lambda calculus is the possibility of type safety and ease of development for a type inference algorithm.

Furthermore, there is a large amount of work into why a quantum programming language should be typed. For a relatively thorough review of these see [8]. An example of a language that uses LDTT and is typed is Proto-Quipper-M [10]. Proto-Quipper is linearly typed, preventing no-cloning. Furthermore, it supports higher-order functions and datatypes of qubits, such as lists, and has built-in categorical semantics [10].

3.4 QT for Quantum Verification

A complete description for how to construct a proof verifier from a TT is far beyond scope. However, we can show examples in the literature that go through this process. A proof verifier essentially takes a proof representation (which corresponds to our quantum program via trinitarianism) and verifies that it is properly constructed. Classical methods are outlined in [6].

An empirical quantum verifier, however, faces problems due to quantum collapse, randomness, and no-cloning. Therefore, verification algorithms will almost certainly have to be compile-time[12].

In [21], Tan et. al. propose MorphQPV, which utilizes the isomorphism structure of quantum programs[13] to create approximations for the output of a program. They claim MorphQPV reduces program executions by 107.9x and improves probability of success by around 3x against five benchmarks [21].

An alternative approach is provided in [5],

[12] in the sense that running the program and measuring is not suitable.

[13] isomorphisms as in the categorical sense, as seen in quantum trinitarianism.

where Bauer-Marquart et. al. propose a symbolic execution system for quantum programs entirely at compile time. They claim their symQV improves on previous systems by an order of magnitude, but will have to shift to approximation for larger number of qubits to be feasible [5].

3.5 QT for Quantum Analysis

Similarly, many approaches have been proposed for automatically determining (or approximating) the complexity of quantum programs. For example in [2], Avanzini et. al. propose the use of a Probabilistic Abstract Reduction System (PARS) based in linear logic in order to bound quantum programs. In doing so, they prove the decidability (though great complexity) of "bounding weakest pre-expectation on quantitative program properties of any mixed classical-quantum program" [2].

3.6 QT for Quantum Topology

Earlier we mentioned that HoTT brings with it an enforcement of topological structure. Indeed, enforcing a topological perspective into quantum computation brings about some new ideas. For example, in [24], Wang explores how topological quantum computation can be used as a paradigm for implementing large-scale quantum computing through topological phases of matter [24]. The report primarily focuses on the physics and theory necessary, but interestingly trinitarianism remains a key tool used[14] [24]. See [9] for more.

3.7 An Alternative Hoare TT (HTT)

Though we have shown that LDTT is compatible with quantum mechanics, it is far from the only compatible LDTT[15] In [20], Singhal proposes Hoare Type Theory (HTT) grounded in Floyd-Hoare logic as a means of abstracting programs. At its core, HTT introduces special Hoare types that can isolate a part of a program from the rest of the language and specify their behavior. They propose additions to HTT to create Quantum HTT (QHTT), which they then show is suitable for programming, type-checking, and verifying quantum programs.

[14] Wang proposes a sort-of trinitarianism between Quantum Topology, Quantum Computing, and Quantum Mechanics in [24].

[15] though it is a very popular one, based on the research we reviewed. See [10].

4 Conclusion

We have examined Linear Dependent Type Theory (LDTT) and its corresponding computational trinitarianism, and why this fixes the problems Homotopy Type Theory (HTT) faces with quantum computation. We then review briefly literature who use similar techniques to advance a variety of quantum disciplines. As such, we note that Quantum Type Theory appears to be a new and very active field[16] and there is much to be learned in all areas regrading Quantum Trinitarianism and its applications.

References

[1] Samson Abramsky. *No-Cloning In Categorical Quantum Mechanics*. 2012. arXiv: 0910.2401 [quant-ph]. URL: https://arxiv.org/abs/0910.2401.

[2] Martin Avanzini et al. *On the Hardness of Analyzing Quantum Programs Quantitatively*. 2023. arXiv: 2312.13657 [cs.LO]. URL: https://arxiv.org/abs/2312.13657.

[3] J. Baez and M. Stay. "Physics, Topology, Logic and Computation: A Rosetta Stone". In: *New Structures for Physics*. Springer Berlin Heidelberg, 2010, pp. 95–172. ISBN: 9783642128219. DOI: 10.1007/978-3-642-12821-9_2. URL: http://dx.doi.org/10.1007/978-3-642-12821-9_2.

[4] Gilles Barthe and Thierry Coquand. "An Introduction to Dependent Type Theory". In: *Applied Semantics*. Ed. by Gilles Barthe et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–41. ISBN: 978-3-540-45699-5.

[16] Most of the citations used are very new.

[5] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. "symQV: Automated Symbolic Verification of Quantum Programs". In: *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*. Lübeck, Germany: Springer-Verlag, 2023, pp. 181–198. ISBN: 978-3-031-27480-0. DOI: 10.1007/978-3-031-27481-7_12. URL: https://doi.org/10.1007/978-3-031-27481-7_12.

[6] Nicolaas Govert de Bruijn. "Type-theoretical checking and philosophy of Mathematics". In: *Twenty Five Years of Constructive Type Theory*. Oxford University Press, Oct. 1998. ISBN: 9780198501275. DOI: 10.1093/oso/9780198501275.003.0005. eprint: https://academic.oup.com/book/0/chapter/355192329/chapter-pdf/43762976/isbn-9780198501275-book-part-5.pdf. URL: https://doi.org/10.1093/oso/9780198501275.003.0005.

[7] Dan Christensen. *Homotopy Type Theory Colloquium 2020*. Accessed: 2024-12-04. 2020. URL: https://jdc.math.uwo.ca/papers/HoTT-colloq-2020.pdf.

[8] Ross Duncan. *Types for Quantum Computation*. Tech. rep. Accessed: 2024-12-06. University of Oxford, 2006. URL: https://www.cs.ox.ac.uk/files/2425/Types%20for%20Quantum%20Computation.pdf.

[9] Michael H. Freedman et al. *Topological Quantum Computation*. 2002. arXiv: quant-ph/0101025 [quant-ph]. URL: https://arxiv.org/abs/quant-ph/0101025.

[10] Peng Fu, Kohei Kishida, and Peter Selinger. "Linear Dependent Type Theory for Quantum Programming Languages". In: *Logical Methods in Computer Science* Volume 18, Issue 3 (2022). ISSN: 1860-5974. DOI: 10.46298/lmcs-18(3:28)2022. URL: http://dx.doi.org/10.46298/lmcs-18(3:28)2022.

[11] Herman Geuvers. *Introduction to Type Theory*. Accessed: 2024-12-06. 2004. URL: https://www.cs.ru.nl/~herman/onderwijs/provingwithCA/paper-lncs.pdf.

[12] Mohsen Heidari and Wojciech Szpankowski. *New Bounds on Quantum Sample Complexity of Measurement Classes*. 2024. arXiv: 2408.12683 [quant-ph]. URL: https://arxiv.org/abs/2408.12683.

[13] Paul-André Melliès. "Categorical Semantics of Linear Logic". In: *Panoramas et syntheses* (2009).

[14] nLab. *No-Cloning Theorem*. 2023. URL: https://ncatlab.org/nlab/show/no-cloning+theorem.

[15] nLab. *Quantum Circuits Via Dependent Linear Types*. 2024. URL: https://ncatlab.org/nlab/show/quantum+circuits+via+dependent+linear+types.

[16] nLab. *Relation between Type Theory and Category Theory*. 2023. URL: https://ncatlab.org/nlab/show/relation+between+type+theory+and+category+theory.

[17] Valeria de Paiva. "Categorical Semantics of Linear Logic for All". In: *Advances in Natural Deduction: A Celebration of Dag Prawitz's Work*. Ed. by Luiz Carlos Pereira, Edward Hermann Haeusler, and Valeria de Paiva. Dordrecht: Springer Netherlands, 2014, pp. 181–192. ISBN: 978-94-007-7548-0. DOI: 10.1007/978-94-007-7548-0_9. URL: https://doi.org/10.1007/978-94-007-7548-0_9.

[18] Mitchell Riley. "A Linear Dependent Type Theory with Identity Types as a Quantum Verification Language". In: 2023. URL: https://ncatlab.org/nlab/files/Riley-QuantumCertification.pdf.

[19] Peter Selinger and Benoit Valiron. "A lambda calculus for quantum computation with classical control". In: *Mathematical. Structures in Comp. Sci.* 16.3 (June 2006), pp. 527–552. ISSN: 0960-1295. DOI: 10.

1017/S0960129506005238. URL: https://doi.org/10.1017/S0960129506005238.

[20] Kartik Singhal. "Quantum Hoare Type Theory". In: *CoRR* abs/2012.02154 (2020). arXiv: 2012.02154. URL: https://arxiv.org/abs/2012.02154.

[21] Siwei Tan et al. "MorphQPV: Exploiting Isomorphism in Quantum Programs to Facilitate Confident Verification". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 671–688. ISBN: 9798400703867. DOI: 10.1145/3620666.3651360. URL: https://doi.org/10.1145/3620666.3651360.

[22] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[23] Philip Wadler. "Linear Types can Change the World!" In: *Programming Concepts and Methods*. 1990. URL: https://api.semanticscholar.org/CorpusID:58535510.

[24] Zhenghan Wang. *Topological Quantum Computation*. Tech. rep. Accessed: 2024-12-06. Microsoft Research Station Q, 2010. URL: https://web.math.ucsb.edu/~zhenghwa/data/course/cbms.pdf.

5 Appendix

Logic	Set Theory (Internal Logic of)	Category Theory	Type Theory
Proposition	Set	Object	Type
Predicate	Family of sets	Display morphism	Dependent type
Proof	Element	Generalized element	Term/Program
Cut rule		Composition of classifying morphisms / Pullback of display maps	Substitution
Introduction rule for implication		Counit for hom-tensor adjunction	Lambda
Elimination rule for implication		Unit for hom-tensor adjunction	Application
Cut elimination for implication		One of the zigzag identities for hom-tensor adjunction	Beta reduction
Identity elimination for implication		The other zigzag identity for hom-tensor adjunction	Eta conversion
True	Singleton	Terminal object/(-2)-truncated object	H-level 0-type/Unit type
False	Empty set	Initial object	Empty type
Proposition, truth value	Subsingleton	Subterminal object/(-1)-truncated object	H-proposition, Mere proposition
Logical conjunction	Cartesian product	Product	Product type
Disjunction	Disjoint union (support of)	Coproduct ((-1)-truncation of)	Sum type (Bracket type of)
Implication	Function set (into subsingleton)	Internal hom (into subterminal object)	Function type (into H-proposition)
Negation	Function set into empty set	Internal hom into initial object	Function type into empty type
Universal quantification	Indexed cartesian product (of family of subsingletons)	Dependent product (of family of subterminal objects)	Dependent product type (of family of H-propositions)
Existential quantification	Indexed disjoint union (support of)	Dependent sum ((-1)-truncation of)	Dependent sum type (Bracket type of)
Logical equivalence	Bijection set	Object of isomorphisms	Equivalence type
Support set	Support object/(-1)-truncation	Propositional truncation/Bracket type	
N-image of morphism into terminal object/N-truncation		N-truncation modality	
Equality	Diagonal function/Diagonal subset/Diagonal relation	Path space object	Identity type/Path type
Completely presented set	Set	Discrete object/0-truncated object	H-level 2-type/Set/H-set
Set	Set with equivalence relation	Internal 0-groupoid	Bishop set/Setoid with its pseudo-equivalence relation an actual equivalence relation
Equivalence class/Quotient set	Quotient		Quotient type
Induction		Colimit	Inductive type, W-type, M-type
Higher induction		Higher colimit	Higher inductive type
-		0-truncated higher colimit	Quotient inductive type
Coinduction		Limit	Coinductive type
Preset			Type without identity types
Set of truth values	Subobject classifier		Type of propositions
Domain of discourse	Universe	Object classifier	Type universe
Modality		Closure operator, (Idempotent) monad	Modal type theory, Monad (in computer science)
Linear logic		(Symmetric, closed) monoidal category	Linear type theory/Quantum computation
Proof net		String diagram	Quantum circuit
(Absence of) contraction rule		(Absence of) diagonal	No-cloning theorem
Synthetic mathematics			Domain specific embedded programming language

Table 2: Logic, Set Theory, Category Theory, and Type Theory from [16]