The Calculus of Linear Constructions

Qiancheng Fu
Boston University
Boston, MA, USA
qcfu@bu.edu

Abstract

The Calculus of Linear Constructions (CLC) is an extension of the Calculus of Constructions (CC) with linear types. Specifically, CLC extends CC with a hierarchy of linear universes that precisely controls the weakening and contraction of its term level inhabitants. We study the meta-theory of CLC, showing that it is a sound logical framework for reasoning about resource. CLC is backwards compatible with CC, allowing CLC to enjoy the fruits of decades of CC research. We have formalized and proven correct all major results of the core calculus in the Coq Proof Assistant. We extend CLC with linear inductive types, showing that CLC facilitates correct by construction imperative programming.

ACM Reference Format:
Qiancheng Fu. 2021. The Calculus of Linear Constructions. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Calculus of Constructions (CC) is a dependent type theory introduced by Coquand and Huet in their landmark work [8]. In CC types can depend on terms, allowing one to write precise specifications as types. Today, CC and its variations CIC [22] and ECC [15] lie at the core of popular proof assistants such as Coq [26], Agda [21], Lean [9] and others. These theorem provers have found great success in the fields of software verification, and constructive mathematics.

However, due to its origins as a logical framework for constructive mathematics, it is quite difficult for CC to encode and reason about resources. Intuitively, a mathematical theorem can be applied an unrestricted number of times. Comparatively, the usage of resources is more limited. For example, if we encode Girard's classical example [12] of purchasing cigarettes literally into CC as a function of type:

$$\mathit{Money} \to \mathit{Camels} + \mathit{Marlboro}$$

If viewed as a propositional implication, the customer will still maintain full ownership of their money after paying the vendor, because implication does not diminish the validity of its antecedent. Unless the vendor is exceedingly generous, we are faced with the crime of counterfeiting. Users of proof assistants based on CC often need to embed external logics such as Separation Logic to provide additional reasoning principles for dealing with resource. The design and embedding of these logics are a difficult problem in their own right, requiring additional proofs to justify their soundness, and effectiveness. We propose an alternative solution: extend CC with linear types.

This paper presents a new linear dependent type system — The Calculus of Linear Constructions (CLC). CLC extends CCω with linear types. CCω itself is an extension of CC with a cumulative hierarchy of type universes. We add an extra universe sort L of linear types with cumulativity parallel to the sort U of non-linear types. This ultimately cumulates in the *linearity* theorem, stating that all resources are used exactly once.

The presence of both linear and dependent types enables CLC to write specifications that faithfully encode the usage of resources. The previous example of monetary transaction can be refined using an indexed linear type family $\mathit{Money} : \mathbb{N} \to L$ as follows.

$$(x :_L \mathit{Money}\ 5) \to (\mathit{Camels} + \mathit{Marlboro})$$

This new specification for the transaction demands a payment of 5 units of money, and the customer be relieved of their ownership after the transaction finishes, effectively preventing the contradiction of having your cake and eating it too.

Compared to preexisting approaches for integrating linear types and dependent types, CLC offers a "lightweight" approach to extending CCω with linear types, akin to Mazurak et al.'s work on System F [17]. Prior works have either utilized a literal ! exponential or a pair of adjoint connectives to bridge the gap between linear and non-linear types, adding clutter that greatly hinders their practical usage. Our method requires neither exponential nor adjoint connectives, allowing for a straightforward modeling of CLC in CCω, and lifting of CCω into CLC with minimal annotations, endowing CLC with the fruits of decades of CC research.

We further extend CLC with inductive types, showing that CLC facilitates correct by construction imperative programming. We have formalized all major results in Coq, and implemented a prototype in OCaml.

Contributions: Our contributions can be summarized as follows.

- Fist, we describe the Calculus of Linear Constructions, an extension to the Calculus of Constructions with linear types. The integration of linear types and dependent types allows CLC to directly and precisely reason about resource.
- Second, we study the meta-theory of CLC directly, showing that it satisfies the standard properties of confluence, propagation and subject reduction.
- Next, we prove the linearity theorem, showing the tangible impact linear types have on the structure of terms. We also show that promotion and dereliction of linear logic are derivable as theorems for canonical types.
- Additionally, we observe that CLC is highly backwards compatible with CCω, allowing us to construct a reduction preserving model of CLC in CCω, showing that CLC is consistent. We also prove that all valid CCω terms after adding minimal amounts of annotation are valid CLC terms.
- All major results have been formalized and proven correct in the Coq Proof Assistant with help from the Autosubst [24] library. To the best of our knowledge, our development is the first machine checked formalization of a linear dependent type theory.
- Furthermore, we extend CLC with linear inductive data, demonstrating CLC's capabilities for correct by construction imperative programming.
- Finally, we give an implementation extended with user definable linear and non-linear inductive types. Algorithmic type checking employed by the implementation streamlines the process of writing CLC.

2 The Language of CLC

2.1 Syntax

Figure 1. Syntax

k	$:=$	$0 \mid 1 \mid 2 \ldots$	predicative levels
i	$:=$	$* \mid 0 \mid 1 \mid 2 \ldots$	all levels
s, t	$::=$	$U \mid L$	sorts
m, n, A, B, C	$::=$	$U_i \mid L_k \mid x$	expressions
	$\mid$	$(x :_s A) \to B$	
	$\mid$	$(x :_s A) \multimap B$	
	$\mid$	$\lambda x.n \mid m\ n$	

The syntax of the core type theory is presented in Figure 1. Our type theory contains two sorts of universes U and L. We use the meta variable k to specifically quantify over levels $0, 1, 2, \ldots$ that correspond to the predicative universes. We use the meta variable i to quantify over all levels $*, 0, 1, 2, \ldots$. Here, U_* is the impredicative universe of propositions, in the same spirit as CCω's *Prop* universe. U_k, and L_k are the predicative universes of non-linear and linear types respectively.

Figure 2. Correspondence of CLC types and MELL implications

$$(\_ :_U A) \to B \quad \Leftrightarrow \quad !(!A \multimap B) \qquad (1)$$
$$(\_ :_L A) \to B \quad \Leftrightarrow \quad !(A \multimap B) \qquad (2)$$
$$(\_ :_U A) \multimap B \quad \Leftrightarrow \quad !A \multimap B \qquad (3)$$
$$(\_ :_L A) \multimap B \quad \Leftrightarrow \quad A \multimap B \qquad (4)$$

A clear departure of our language from standard presentations of both linear type theory and dependent type theory is the presence of two function types: $(x :_s A) \to B$, $(x :_s A) \multimap B$. The reason for these function types is that we have built the ! exponential of linear logic directly into universe sorts. The sort annotation s here records the universe sort of the function domain. The behavior of ! is difficult to account for even in simple linear type theories. Subtle issues arise if !! is not canonically isomorphic to !, which may invalidate the substitution lemma [30]. By integrating the exponential directly into universe sorts, we implicitly limit ! to only be canonically usable. This allows us to derive the substitution lemma, and construct a direct modeling of CLC in CCω without requiring any additional machinery for manipulating exponential.

Figure 2 illustrates the correspondence between CLC function types and Multiplicative Exponential Linear Logic (MELL) implications. MELL lacks counterparts for the cases (1), (3) if the co-domain B is dependent on arguments of domain A. We will discuss function type formation in Section 2.5 in greater detail.

In practice, algorithmic type checking techniques allow users to omit writing sort indices. Our implementation employs bi-directional type checking and users never interact with indices in the surface syntax.

2.2 Universes and Cumulativity

CLC features two sorts of universes U and L with level indices $*, 0, 1, 2, \cdots$. U_* is the impredicative universe of propositions. U_k and L_k are the predicative universes of non-linear types and linear types respectively. The main mechanism that CLC uses to distinguish between linear and non-linear types is by checking the universe to which they belong. Basically, terms with types that occur within U_i are unrestricted in their usage. Terms with types that occur within L_k are restricted to being used exactly once.

In order to lift terms from lower universes to higher ones, there exists cumulativity between universe levels of the same sort. We define cumulativity as follows.

Definition 2.1. The cumulativity relation $(\preceq)$ is the smallest binary relation over terms such that

1. $\preceq$ is a partial order with respect to equality.

   a. If $A \equiv B$, then $A \preceq B$.
   b. If $A \preceq B$ and $B \preceq A$, then $A \equiv B$.
   c. If $A \preceq B$ and $B \preceq C$, then $A \preceq B$.
2. $U_* \preceq U_0 \preceq U_1 \preceq U_2 \preceq \cdots$
3. $L_0 \preceq L_1 \preceq L_2 \preceq \cdots$
4. If $A_1 \equiv A_2$ and $B_1 \preceq B_2$,
   then $(x :_s A_1) \to B_1 \preceq (x :_s A_2) \to B_2$
5. If $A_1 \equiv A_2$ and $B_1 \preceq B_2$,
   then $(x :_s A_1) \multimap B_1 \preceq (x :_s A_2) \multimap B_2$

Figure 3 illustrates the structure of our universe hierarchy. Each linear universe L_k has U_{k+1} as its type, allowing functions to freely quantify over linear *types*. However, L_k cumulates to L_{k+1}. These two parallel threads of cumulativity prevent linear types from being transported to the non-linear universes, and subsequently losing track of linearity.

Figure 3. The Universe Hierarchy

$$\begin{tikzcd}
U_* \arrow[r, dash, "\preceq", ":"'] & U_0 \arrow[r, dash, "\preceq", ":"'] & U_1 \arrow[r, dash, "\preceq", ":"'] & U_2 \arrow[r, dash, "\preceq", ":"'] & \cdots \\
& L_0 \arrow[ur, dash, ":"] \arrow[r, dash, "\preceq"] & L_1 \arrow[ur, dash, ":"] \arrow[r, dash, "\preceq"] & L_2 \arrow[ur, dash, ":"] \arrow[r, dash, "\preceq"] & \cdots
\end{tikzcd}$$

2.3 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[16]. Variables in the context are annotated to indicate whether they are linear or non-linear. A non-linear variable is annotated as $\Gamma, x :_U A$, whereas a linear variable is annotated as $\Gamma, x :_L A$.

Next, we define a $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$ relation that merges two mixed contexts Γ_1, and Γ_2 into Γ, by performing contraction on shared non-linear variables. For linear variables, the $\_ \ddagger \_ \ddagger \_$ relation is defined if and only if each variable occurs uniquely in one context and not the other. This definition of $\_ \ddagger \_ \ddagger \_$ is what allows contraction for unrestricted variables whilst forbidding it for restricted ones.

An auxiliary judgment $|\Gamma|$ is defined to assert that a context Γ does not contain linear variables. In other words, all variables found in $|\Gamma|$ are annotated of the form $x :_U A$. The full rules for structural judgments are presented in Figure 4.

Definition 2.2. The context restriction function $\overline{\Gamma}$ is defined as a recursive filter over Γ as follows. All linear variables are removed from context Γ. The result of context restriction is the non-linear subset of the original context.

$$\overline{\epsilon} = \epsilon \qquad \overline{\Gamma, x :_U A} = \overline{\Gamma}, x :_U A \qquad \overline{\Gamma, x :_L A} = \overline{\Gamma}$$

2.4 Typing Judgment

Typing judgments in CLC take on the form of $\Gamma \vdash m : A$. Intuitively, this judgment states that the term m is an inhabitant of type A, with free variables typed in Γ.

Definition 2.3. We formally define the terms *non-linear*, *linear*, *unrestricted* and *restricted.*

Figure 4. Structural Judgments

$$\frac{}{\epsilon \vdash}\textsc{Wf-}\epsilon \qquad \frac{\Gamma \vdash \qquad \overline{\Gamma} \vdash A :_U U_i}{\Gamma, x :_U A \vdash}\textsc{Wf-U}$$

$$\frac{\Gamma \vdash \qquad \overline{\Gamma} \vdash A :_U L_k}{\Gamma, x :_L A \vdash}\textsc{Wf-L}$$

$$\frac{}{|\epsilon|}\textsc{Pure-}\epsilon \qquad \frac{|\Gamma| \qquad \Gamma \vdash A :_U U}{|\Gamma, x :_U A|}\textsc{Pure-U}$$

$$\frac{}{\epsilon \ddagger \epsilon \ddagger \epsilon}\textsc{Merge-}\epsilon$$

$$\frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma_1, x :_U A \ddagger \Gamma_2, x :_U A \ddagger \Gamma, x :_U A}\textsc{Merge-U}$$

$$\frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma \qquad x \notin \Gamma_2}{\Gamma_1, x :_L A \ddagger \Gamma_2 \ddagger \Gamma, x :_L A}\textsc{Merge-L1}$$

$$\frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma \qquad x \notin \Gamma_1}{\Gamma_1 \ddagger \Gamma_2, x :_L A \ddagger \Gamma, x :_L A}\textsc{Merge-L2}$$

1. A *type* A is *non-linear* under context Γ if $\Gamma \vdash A : U_i$.
2. A *type* A is *linear* under context Γ if $\Gamma \vdash A : L_k$.
3. A *term* m is *unrestricted* under context Γ if there exists non-linear type A, such that $\Gamma \vdash m : A$. A unrestricted term may be used an arbitrary number of times.
4. A *term* m is *restricted* under context Γ if there exists linear type A, such that $\Gamma \vdash m : A$. A restricted term must be used exactly once.

2.5 Type Formation

The rules for forming types are presented in Figure 5. In CLC, we forbid types from depending on linear terms similar to [7, 14] for the same reason of avoiding philosophical troubles.

The axiom rules Prop-Axiom, U-Axiom, L-Axiom are almost standard, the main difference being the extra side-condition of judgment $|\Gamma|$. In most presentations of dependent type theories without linear types, the universe axioms are derivable under any well-formed context Γ. Variables not pertaining to actual proofs could be introduced this way, thus giving rise to the admissibility of weakening. To support linear types, we must restrict weakening to non-linear variables. This justifies the restriction of Γ to contain only non-linear variables for Prop-Axiom, U-Axiom, L-Axiom. From L-Axiom we can see that the universe of linear types L_k is an inhabitant of U_{k+1}. This is inspired by Krishnaswami et al.'s treatment of linear universes [14], where linear *types* themselves can be used unrestrictedly.

The Prop rule is used for forming propositions. From the judgment $\Gamma \vdash A : U_i$ we can see that Prop allows impredicative quantification over non-linear types of arbitrary level i. The judgment $\Gamma, x :_U A \vdash B : U_*$ asserts that the co-domain must be in the impredicative universe U_*. The final resulting judgment $\Gamma \vdash (x :_U A) \to B : U_*$ has sort U, indicating that terms with type $(x :_U A) \to B$ enjoy unrestricted usage.

Figure 5. Type Formation

$$\frac{|\Gamma|}{\Gamma \vdash U_* : U_0}\textsc{Prop-Axiom} \qquad \frac{|\Gamma|}{\Gamma \vdash s_k : U_{k+1}}\textsc{Sort-Axiom}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma, x :_U A \vdash B : U_*}{\Gamma \vdash (x :_U A) \to B : U_*}\textsc{Prop}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_k \qquad \Gamma, x :_U A \vdash B : s_k}{\Gamma \vdash (x :_U A) \to B : U_k}\textsc{U-Prod}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_k \qquad \Gamma \vdash B : s_k \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \to B : U_k}\textsc{L-Prod}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_k \qquad \Gamma, x :_U A \vdash B : s_k}{\Gamma \vdash (x :_U A) \multimap B : L_k}\textsc{U-Lolli}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_k \qquad \Gamma \vdash B : s_k \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \multimap B : L_k}\textsc{L-Lolli}$$

The U-Prod rule is used for forming non-linear function types with non-linear domains. This is evident from the judgment $\Gamma \vdash A : U_k$. Due to the fact that A is in the predicative non-linear universe U_k, terms of type A have unrestricted usage. The non-linearity of domain A allows B to depend on terms of type A, as seen in judgment $\Gamma, x :_U A \vdash B : s_k$. The domain B itself may be non-linear or linear, since B's universe s_k can vary between U_k and L_k. From the resulting judgment $\Gamma \vdash (x :_U A) \to B : U_k$, we see that the overall type is a non-linear type. The term level λ-abstractions of these types can be used unrestrictedly.

The L-Prod rule is used for forming non-linear function types with linear domains. From the judgment $\Gamma \vdash A : L_k$, we see that A is a linear type, and terms of type A have restricted usage. Because of this, we forbid co-domain B from depending on terms of type A, evident in the judgment $\Gamma \vdash B : s_k$. Like U-Prod, co-domain B itself may be non-linear or linear, since B's universe s_k can vary between U_k and L_k. The final resulting judgment is $\Gamma \vdash (x :_L A) \to B : U_k$. Here, x is a hypocritical unbinding variable whose only purpose is to preserve syntax uniformity. Again, the overall resulting type is non-linear, so the term level λ-abstractions of these types can be used unrestrictedly.

The U-Lolli, L-Lolli rules are similar to U-Prod, L-Prod in spirit. The dependency considerations for domain A and co-domain B are exactly the same. The main difference between U-Lolli, L-Lolli and U-Prod, L-Prod is the universe sort of the resulting type. The sorts of the function types formed by U-Lolli, L-Lolli are L, meaning they are linear function types. The term level λ-abstractions of these types must be used exactly once.

2.6 Term Formation

The rules for term formation are presented in Figure 6.

Figure 6. Term Formation

$$\frac{|\Gamma_1, \Gamma_2|}{\Gamma_1, x :_U A, \Gamma_2 \vdash x : A}\textsc{U-Var} \qquad \frac{|\Gamma_1, \Gamma_2|}{\Gamma_1, x :_L A, \Gamma_2 \vdash x : A}\textsc{L-Var}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash (x :_s A) \to B : t_i \qquad \Gamma, x :_s A \vdash n : B}{\Gamma \vdash \lambda x.n : (x :_s A) \to B}\textsc{Prod-}\lambda$$

$$\frac{\overline{\Gamma} \vdash (x :_s A) \multimap B : t_i \qquad \Gamma, x :_s A \vdash n : B}{\Gamma \vdash \lambda x.n : (x :_s A) \multimap B}\textsc{Lolli-}\lambda$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : (x :_U A) \to B \\ |\Gamma_2| \qquad \Gamma_2 \vdash n : A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\end{array}}{\Gamma \vdash m\ n : B[n/x]}\textsc{U-Prod-App}$$

$$\frac{\Gamma_1 \vdash m : (x :_L A) \to B \qquad \Gamma_2 \vdash n : A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n : B[n/x]}\textsc{L-Prod-App}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : (x :_U A) \multimap B \\ |\Gamma_2| \qquad \Gamma_2 \vdash n : A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\end{array}}{\Gamma \vdash m\ n : B[n/x]}\textsc{U-Lolli-App}$$

$$\frac{\Gamma_1 \vdash m : (x :_L A) \multimap B \qquad \Gamma_2 \vdash n : A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n : B[n/x]}\textsc{L-Lolli-App}$$

$$\frac{\Gamma \vdash m : A \qquad \overline{\Gamma} \vdash B : s_i \qquad A \preceq B}{\Gamma \vdash m : B}\textsc{Conv}$$

The rules U-Var, and L-Var are used for typing free variables. The U-Var rule asserts that free variable x occurs within the context $\Gamma_1, x :_U A, \Gamma_2$ with a non-linear type A. The L-Var rule asserts that free variable x occurs within the context $\Gamma_1, x :_L A, \Gamma_2$ with linear type A. For both rules, the side condition $|\Gamma_1, \Gamma_2|$ forbids irrelevant variables of linear types from occurring within the context. This prevents another vector of weakening variables with linear type.

In Prod-λ, the function type being addressed has form $(x :_s A) \to B$. λ-abstractions of this type can be applied an unrestricted number of times, hence cannot depend on free variables with restricted usage without possibly duplicating them. This consideration is realized by the side condition $|\Gamma|$, asserting all variables in context Γ are unrestricted. Next, the body of the abstraction n is typed as $\Gamma, x :_s A \vdash n : B$, where s is the sort of A. Finally, the resulting judgment $\Gamma \vdash \lambda x.n : (x :_s A) \rightarrow B$ asserts that the λ-abstraction can be used unrestrictedly.

In contrast to Prod-λ, the Lolli-λ rule is used for forming λ-abstractions that must be used exactly once. Due to the fact these abstractions must be used once, they are allowed access to restricted variables within context Γ, evident in the judgment $\Gamma, x :_s A \vdash n : B$, and lack of side condition $|\Gamma|$. However, in judgment $\overline{\Gamma} \vdash (x :_s A) \multimap B : L_k$ the context must be filtered, because types are not allowed to depend on restricted variables. The final resulting judgment $\Gamma \vdash \lambda x.n : (x :_s A) \multimap B$ asserts that the λ-abstraction must be used exactly once.

For U-Prod-App, domain A is a non-linear type, as seen by its annotation in $(x :_U A) \rightarrow B$. Intuitively, this tells us that x may be used an arbitrary number of times within the body of m. Thus, the supplied argument n must not depend on restricted variables in context Γ_2. Otherwise, substitution may put multiple copies of n into m during β-reduction, duplicating variables that should have been restricted. This justifies the side condition of $|\Gamma_2|$. The contexts Γ_1, and Γ_2 are finally merged together into Γ by the relation $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$, contracting all unrestricted variables shared between Γ_1, and Γ_2.

Now for L-Prod-App, domain A is a linear type, as seen by its annotation in $(x :_L A) \rightarrow B$. Intuitively, this tells us that x must be used once within the body of m. During β-reduction, substitution will only put a single copy of n into the body of m, so n can depend on restricted variables within Γ_2 without fear of duplicating them. This justifies the lack of side condition $|\Gamma_2|$. The contexts Γ_1, and Γ_2 are finally merged together into Γ by the relation $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$, contracting all unrestricted variables shared between Γ_1, and Γ_2.

The rules U-Lolli-App, L-Lolli-App follow the same considerations as U-Prod-App, L-Prod-App. Additional conditions are not required for these two rules to be sound.

Finally, the Conv rule allows judgment $\Gamma \vdash m : A$ to convert to judgment $\Gamma \vdash m : B$ if B is a valid type in context $\overline{\Gamma}$. Futhermore, A must be a subtype of B satisfying the cumulativity relation $A \preceq B$. This rule gives rise to large eliminations (compute types from terms), as computations embedded at the type level can convert to canonical types.

2.7 Equality and Reduction

The operational semantics and definitional β-equality of CLC are presented in Figure 7, all of which are entirely standard. As we have discussed previously, the elimination of the explicit ! exponential allows CLC to maintain the standard operational semantics of CCω, whose β-reductions are very well behaved.

Figure 7. Equality and Parallel Reduction

$$\frac{m_1 \leadsto^* n \qquad m_2 \leadsto^* n}{m_1 \equiv m_2 : A}\textsc{Join} \qquad \frac{}{(\lambda x.m)\ n \leadsto m[n/x]}\textsc{P-}\beta$$

$$\frac{m \leadsto m'}{\lambda x.m \leadsto \lambda x.m'}\textsc{Step-}\lambda$$

$$\frac{A \leadsto A'}{(x :_s A) \rightarrow B \leadsto (x :_s A') \rightarrow B}\textsc{Step-ProdL}$$

$$\frac{B \leadsto B'}{(x :_s A) \rightarrow B \leadsto (x :_s A) \rightarrow B'}\textsc{Step-ProdR}$$

$$\frac{A \leadsto A'}{(x :_s A) \multimap B \leadsto (x :_s A') \multimap B}\textsc{Step-LolliL}$$

$$\frac{B \leadsto B'}{(x :_s A) \multimap B \leadsto (x :_s A) \multimap B'}\textsc{Step-LolliR}$$

$$\frac{m \leadsto m'}{m\ n \leadsto m'\ n}\textsc{Step-AppL} \qquad \frac{n \leadsto n'}{m\ n \leadsto m\ n'}\textsc{Step-AppR}$$

2.8 Meta Theory

In this section, we focus our discussion on the properties of CLC. First, we show the type soundness of CLC through the *subject reduction* theorem. Next, we show that CLC is a valid linear type theory through the *linearity* theorem. Furthermore, we show that the *promotion* and *dereliction* rules can be encoded as η-expansions. Finally, we construct a reduction preserving erasure function that maps well-typed CLC terms to well-typed CCω terms, showing that CLC is strongly normalizing.

All proofs have been formalized in Coq with help from the Autosubst [24] library. The Coq development is publicly available on the first author's Github repository. To the best of our knowledge, this is the first machined checked formalization of a linear dependently type theory. We give a hand written version of the proof in the appendix as well.

2.8.1 Reduction and Confluence.

The following lemmas and proofs are entirely standard using parallel step technique [25]. The presence of linear types do not pose any complications as reductions are untyped.

Lemma 2.1. *Parallel reduction satisfies the diamond property. If $m \leadsto_p m_1$ and $m \leadsto_p m_2$ then there exists m' such that $m_1 \leadsto_p m'$ and $m_2 \leadsto_p m'$.*

Lemma 2.2. *Each $\rightsquigarrow_p$ is reachable with $\rightsquigarrow^*$. If $m \rightsquigarrow_p m'$ then $m \rightsquigarrow^* m'$.*

Theorem 2.3. *The transitive reflexive closure of reduction is confluent. If $m \rightsquigarrow^* m_1$ and $m \rightsquigarrow^* m_2$ then there exists m' such that $m_1 \rightsquigarrow^* m'$ and $m_2 \rightsquigarrow^* m'$.*

Corollary 2.3.1. *The definitional equality relation $\equiv$ is an equivalence relation.*

2.8.2 Weakening.

CLC restricts the weakening rule for variables of linear types. However, weakening variables of non-linear types remain admissible.

Lemma 2.4. *Weakening. If $\Gamma \vdash m : A$ is a valid judgment, then for any $x \notin \Gamma$, the judgment $\Gamma, x :_U B \vdash m : A$ is derivable.*

2.8.3 Substitution.

Though the substitution lemma is regarded as a boring and bureaucratic result, it is surprisingly hard to design linear typed languages where the substitution lemma is admissible. Much of the difficulty arises during the substitution of arguments containing ! exponential. Perhaps the most famous work detailing the issues of substitution is due to Wadler [30]. He defines additional syntax and semantics for the intricate unboxing of ! terms, solving the lack of substitute in Abramsky's term calculus.

Our design of integrating ! into universe sorts removes the need for ! manipulating syntax and semantics. The following substitution lemmas are directly proved by induction on typing derivations.

Lemma 2.5. *Non-linear Substitution. For $\Gamma_1, x :_U A \vdash m : B$ and $\Gamma_2 \vdash n : A$, if $|\Gamma_2|$ and $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$ are valid for some Γ, then $\Gamma \vdash m[n/x] : B[n/x]$.*

Lemma 2.6. *Linear Substitution. For $\Gamma_1, x :_L A \vdash m : B$ and $\Gamma_2 \vdash n : A$, if $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$ is valid for some Γ, then $\Gamma \vdash m[n/x] : B[n/x]$.*

2.8.4 Type Soundness.

In order to prove subject reduction, we first prove the propagation theorem. The main purpose of propagation is to lower types down to the term level, enabling the application of various inversion lemmas.

Theorem 2.7. *Propagation. For any context Γ, term m, type A, and sort s, if $\Gamma \vdash m : A$ is a valid judgment, then there exist some sort s and level i such that $\overline{\Gamma} \vdash A : s_i$.*

With weakening, substitution, propagation and various inversion lemmas proven, subject reduction can now be proved by induction on typing derivation.

Theorem 2.8. *Subject Reduction. For $\Gamma \vdash m : A$, if $m \rightsquigarrow n$ then $\Gamma \vdash n : A$.*

2.8.5 Linearity.

At this point, we have proven that CLC is type sound. However, we still need to prove that the removal of weakening and contraction for restricted variables yields tangible impact on the structure of terms. For this purpose, we define a binding aware recursive function $occurs(x, m)$ that counts the number of times variable x appears within term m. The linearity theorem asserts that restricted variables are used exactly once within a term, subsuming safe resource usage.

Before we proceed, we first prove the seemingly obvious narity lemma.

Lemma 2.9. *Narity. If $\Gamma \vdash m : A$ is a valid judgment, for any $x \notin \Gamma$, there is $occurs(x, m) = 0$.*

For cases with branched syntax in the linearity theorem, such as application $(m\ n)$, the *occurs* function sums up the occurrences of variable x in branches m and n. The narity lemma is used to prove that either $occurs(x, m) = 1 \land occurs(x, n) = 0$ or $occurs(x, m) = 0 \land occurs(x, n) = 1$ is true.

Theorem 2.10. *Linearity. If $\Gamma \vdash m : A$ is a valid judgment, for any $(x :_L B) \in \Gamma$, there is $occurs(x, m) = 1$.*

2.8.6 Promotion and Dereliction.

Linear types of CLC are not obtained through packing and unpacking !, so there are no explicit rules for *promotion* and *dereliction* of Linear Logic. Arbitrary computations existing at the type level also muddle the association between which types can be promoted or derelicted to which. Nevertheless, *promotion* and *dereliction* for canonical types are derivable as theorems through η-expansion.

Theorem 2.11. *Promotion. If $\Gamma \vdash m : (x :_s A) \multimap B$ and $|\Gamma|$ are valid judgments, then there exists n such that $\Gamma \vdash n : (x :_s A) \to B$ is derivable.*

Theorem 2.12. *Dereliction. If $\Gamma \vdash m : (x :_s A) \to B$ is a valid judgment, then there exists n such that $\Gamma \vdash n : (x :_s A) \multimap B$ is derivable.*

2.8.7 Strong Normalization.

The strong normalization theorem of CLC is proven by construction of a typing and reduction preserving erasure function to CCω. We assume familiarity with CCω syntax here, and define the erasure function as follows.

Definition 2.4.

$$\begin{aligned}
[\![x]\!] &= x\\
[\![U_*]\!] &= Prop\\
[\![U_k]\!] &= Type_k\\
[\![L_k]\!] &= Type_k\\
[\![(x :_s A) \to B]\!] &= (x : [\![A]\!]) \to [\![B]\!]\\
[\![(x :_s A) \multimap B]\!] &= (x : [\![A]\!]) \to [\![B]\!]\\
[\![\lambda x.n]\!] &= \lambda x.[\![n]\!]\\
[\![m\ n]\!] &= [\![m]\!]\ [\![n]\!]
\end{aligned}$$

With slight overloading of notation, we define erasure for CLC contexts recursively.

Definition 2.5.

$$[\![\epsilon]\!] = \epsilon$$
$$[\![\Gamma, x :_s A]\!] = [\![\Gamma]\!], x : [\![A]\!]$$

We prove the following lemma to commute erasure and substitution whenever needed.

Lemma 2.13. *Erasure commutes with substitution. For any CLC terms m, n and some variable x, $[\![m[n/x]]\!] = [\![m]\!][[\![n]\!]/x]$.*

For the following lemma, we refer to the reductions in CLC as $\rightsquigarrow_{\text{CLC}}$, and the reductions in CCω as $\rightsquigarrow_{\text{CC}\omega}$.

Theorem 2.14. *Erasure preserves reduction. For any CLC terms m and n, if there is $m \rightsquigarrow_{CLC} n$, then there is $[\![m]\!] \rightsquigarrow_{CC\omega} [\![n]\!]$.*

Theorem 2.15. *Embedding. If $\Gamma \vdash m :_s A$ is a valid judgment in CLC, then $[\![\Gamma]\!] \vdash [\![m]\!] : [\![A]\!]$ is a valid judgment in CCω.*

If there exists some well typed CLC term with an infinite sequence of reductions, erasure will embed this term into a well typed CCω term along with its infinite sequence of reductions by virtue of theorems 2.14, and 2.15. This is contradictory to the strong normalization property of CCω proven through the Girard-Tait method [15], so this hypothetical term does not exist in CLC.

Theorem 2.16. *Well-typed CLC terms are strongly normalizing.*

2.8.8 Backwards Compatibility. To show that CLC is backwards compatible with CCω, we construct a function that annotates CCω terms with sort U, lifting them into the non-linear fragment of CLC.

Definition 2.6.

$$\begin{aligned} (\!|x|\!) &= x \\ (\!|Prop|\!) &= U_* \\ (\!|Type_k|\!) &= U_k \\ (\!|(x : A) \to B|\!) &= (x :_U (\!|A|\!)) \to (\!|B|\!) \\ (\!|\lambda x.n|\!) &= \lambda x.(\!|n|\!) \\ (\!|m\ n|\!) &= (\!|m|\!)\ (\!|n|\!) \end{aligned}$$

With slight overloading of notation, we define lifting for CCω recursively.

Definition 2.7.

$$(\!|\epsilon|\!) = \epsilon$$
$$(\!|\Gamma, x : A|\!) = (\!|\Gamma|\!), x :_U (\!|A|\!)$$

The following lemmas and theorems are all proved following a similar procedure to proving the soundness of erasure.

Lemma 2.17. *Lift commutes with substitution. For any CCω terms m, n and some variable x, $(\!|m[n/x]|\!) = (\!|m|\!)[(\!|n|\!)/x]$.*

Theorem 2.18. *Lift preserves reduction. For any CCω term m and n, if there is $m \rightsquigarrow_{CC\omega} n$, then there is $(\!|m|\!) \rightsquigarrow_{CLC} (\!|n|\!)$.*

Theorem 2.19. *Lifting. If $\Gamma \vdash m : A$ is a valid judgment in CCω, then $(\!|\Gamma|\!) \vdash (\!|m|\!) : (\!|A|\!)$ is a valid judgment in CLC.*

The lifting theorem shows that all valid CCω terms are valid CLC terms if the domain of function types are annotated with sort U. In practice, even these annotations can be omitted if algorithmic type checking is able to infer them from context.

3 Extensions

Readers familiar with the literature of linear logic or linear type theory will have noticed at this point that CLC does not possess the iconic ⊗ connective. Though it is entirely possible to encode ⊗ in core CLC through a linear Church encoding, proofs and programs written in this style will be incredibly difficult to maintain.

Figure 8. Inductive Type Definition and Elimination

```
(* Sigma Definition *)
Inductive Sigma (A: U) (F: A → U): U :=
| pair : (x: A) → F x → Sigma A F.

(* Tensor Definition *)
Inductive Tensor (A: L) (B: L): L :=
| tpair: A → B → Tensor A B.

(* FTensor Definition *)
Inductive FTensor (A: U) (F: A → L): L :=
| fpair: (x: A) → F x → FTensor A F.

(* Sigma Elimination *)
match m with
| pair x y ⇒ (* clause body *)
end

(* Tensor Elimination *)
match m with
| tpair x y ⇒ (* clause body *)
end

(* FTensor Elimination *)
match m with
| fpair x y ⇒ (* clause body *)
end
```

A major design goal of CLC is to support an ergonomic interface for users to define inductive types [10, 22] as an extension to the core theory. The presence of universe sorts U and L allow users to declare inductive types to be non-linear or linear. The lack of ! exponential simplifies the construction and elimination of inductive terms through direct use of constructors and pattern-matching. The ⊗ connective becomes definable as an inductive type.

Figure 8 is an excerpt taken from our implementation's prelude, with syntax heavily inspired by Coq.

- `Sigma` defines the standard dependent sum type. Since first component of the pair constructor is of non-linear type, the type of the second component is allowed to depend on the first component.
- `Tensor` defines the ⊗ connective for CLC. Both components of the `tpair` constructor are of linear types. When a `Tensor` term is eliminated through pattern-matching, its first and second components must be used exactly once in the body of its pattern clause.
- `FTensor` fulfills a role similar to Krishnaswami et al.'s F adjoint connective [14]. The type of `FTensor`'s first component is non-linear. The linear type of its second component is allowed to depend on the first component. Due to the fact that `FTensor`'s second component is linear, the entire `FTensor` must be linear as well, otherwise duplication of restricted variables may occur.

Due to the complexity of checking soundness for dependent linear type definitions (such as `FTensor`) and pattern-matching based elimination, we have not verified the algorithm employed by our implementation.

Figure 9, Figure 10 and Figure 11 presents the formation, introduction and elimination rules for select inductive types respectively. We will use these inductive types during the construction of examples.

Figure 9. Select Formation Rules

$$\frac{|\Gamma|}{\Gamma \vdash \text{unit} : U_0}\textsc{Unit-Form} \qquad \frac{|\Gamma|}{\Gamma \vdash \mathbb{N} : U_0}\mathbb{N}\textsc{-Form}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma \vdash m : A \qquad \Gamma \vdash n : A}{\Gamma \vdash m =_A n : U_*}{=}\textsc{-Form}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_k \qquad \Gamma x :_U A \vdash B : U_k}{\Gamma \vdash \Sigma x : A.B : U_k}\Sigma\textsc{-Form}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_k \qquad \Gamma \vdash B : L_k}{\Gamma \vdash A \otimes B : L_k}\otimes\textsc{-Form}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_k \qquad \Gamma x :_U A \vdash B : L_k}{\Gamma \vdash Fx : A.B : L_k}F\textsc{-Form}$$

Figure 10. Select Introduction Rules

$$\frac{|\Gamma|}{\Gamma \vdash () : \text{unit}}\textsc{Unit-Intro} \qquad \frac{|\Gamma|}{\Gamma \vdash O : \mathbb{N}}O\textsc{-Intro}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash S\ n : \mathbb{N}}S\textsc{-Intro}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma \vdash m : A}{\Gamma \vdash \text{refl}_A m : m =_A m}{=}\textsc{-Intro}$$

$$\frac{\begin{array}{c}|\Gamma| \qquad \Gamma \vdash m : A \qquad \Gamma \vdash n : B[m/x]\\ \Gamma \vdash \Sigma x : A.B : U_k\end{array}}{\Gamma \vdash (m, n) : \Sigma x : A.B}\Sigma\textsc{-Intro}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : A \qquad \Gamma_2 \vdash n : B \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\\ \Gamma \vdash A \otimes B : L_k\end{array}}{\Gamma \vdash [m, n] : A \otimes B}\otimes\textsc{-Intro}$$

$$\frac{\begin{array}{c}|\Gamma_1| \qquad \Gamma_1 \vdash m : A \qquad \Gamma_2 \vdash n : B[m/x]\\ \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma \qquad \Gamma \vdash Fx : A.B : L_k\end{array}}{\Gamma \vdash \{m, n\} : Fx : A.B}F\textsc{-Intro}$$

Figure 11. Select Elimination Rules

$$\frac{\Gamma_1 \vdash m : \text{unit} \qquad \Gamma_2 \vdash n : A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash \text{let } () := \text{m in n} : A}\textsc{Unit-Elim}$$

$$\frac{|\Gamma|}{\begin{array}{c}\Gamma \vdash \text{iter}^U_{\mathbb{N}} : (C :_U \mathbb{N} \to U_i) \to C\ O \to\\ ((n :_U \mathbb{N}) \to C\ n \to C\ (S\ n)) \to (n :_U \mathbb{N}) \to C\ n\end{array}}\mathbb{N}\textsc{-U-Elim}$$

$$\frac{|\Gamma|}{\begin{array}{c}\Gamma \vdash \text{iter}^L_{\mathbb{N}} : (C :_U \mathbb{N} \to L_k) \to C\ O \multimap\\ ((n :_U \mathbb{N}) \to C\ n \to C\ (S\ n)) \multimap (n :_U \mathbb{N}) \multimap C\ n\end{array}}\mathbb{N}\textsc{-L-Elim}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : \Sigma x : A.B \qquad \Gamma_2, x :_U A, y :_U B \vdash n : C\\ \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\end{array}}{\Gamma \vdash \text{let } (x, y) := m \text{ in } n \vdash C}\Sigma\textsc{-Elim}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : A \otimes B \qquad \Gamma_2, x :_L A, y :_L B \vdash n : C\\ \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\end{array}}{\Gamma \vdash \text{let } [x, y] := m \text{ in } n \vdash C}\otimes\textsc{-Elim}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash m : Fx : A.B \qquad \Gamma_2, x :_U A, y :_L B \vdash n : C\\ \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma\end{array}}{\Gamma \vdash \text{let } \{x, y\} := m \text{ in } n \vdash C}F\textsc{-Elim}$$

Of all the rules presented here, perhaps the most surprising to readers familiar with linear types or dependent types is the $\mathbb{N}$-L-Elim rule. This rule constructs a term with linear type through a primitive recursion principle. The $\multimap$ arrows protect the O-case element of linear type $C\ O$ from duplication by currying. The S-case function is of non-linear type $(n : \mathbb{N}) \to C\ n \to C\ (S\ n)$ because it must be applied repeatedly during iteration.

4 Applications

In this section, we demonstrate some of the possible applications of CLC through examples. We collapse the sort annotations on arrows $(x :_U A) \to B$ and $(x :_L A) \to B$ into $(x : A) \to B$ whenever A's sort is inferable from context. We also omit writing universe levels for the sake of clarity.

4.1 Stateful Reasoning

We first postulate five state axioms. These axioms could be viewed as an interface to a trusted memory allocator.

Definition 4.1. State axioms.

- $l \mapsto_A m : \mathbb{N} \to (A : U) \to A \to L$
  The $\mapsto$ is a linear type constructor. It fulfills a role similar to L3's [19] memory access capability, or Separation Logic's [23] points-to assertion. Intuitively, it is a proof that a term m of non-linear type A is currently stored at address l.
- **new** : $(A : U) \to (m : A) \to Fl : \mathbb{N}.(l \mapsto_A m)$
  new is a function that allocates a new memory cell for term m of non-linear type A. It results in an address l paired with an assertion $(l \mapsto_A m)$.
- **free** : $(A : U) \to Fl : \mathbb{N}.Fm : A.(l \mapsto_A m) \to \text{unit}$
  free is a function that de-allocates the memory cell at location l which is currently in use. This can be seen in its input $Fl : \mathbb{N}.Fm : A.(l \mapsto_A m)$, where the assertion component $(l \mapsto_A m)$ is a proof that location l is currently allocated.
- **get** : $(A : U) \to (l : \mathbb{N}) \to (m : A) \to (l \mapsto_A m) \to F\_ : (\Sigma x.x =_A m).(l \mapsto_A m)$
  get is a function that retrieves the term stored at address l when given a proof $(l \mapsto_A m)$ that there is indeed a term m currently stored there. The type of get's output is the rather complicated $F\_ : (\Sigma x.x =_A m).(l \mapsto_A m)$. This not only returns the term stored at l, but also a proof that the returned term is exactly the same as what $(l \mapsto_A m)$ claimed to have stored. The proof $(l \mapsto_A m)$ is returned unchanged as well.
- **set** : $(A\ B : U) \to (l : \mathbb{N}) \to (m : A) \to (l \mapsto_A m) \multimap (n : B) \multimap (l \mapsto_B n)$
  set has the most complicated type of all the axioms presented here. Intuitively, if address l is allocated and storing some term m of type A, set may put term n of type B into the store at address l, overwriting the original term. This is the so called "strong update" operation. The consumption of $(l \mapsto_A m)$ and return of $(l \mapsto_B n)$ accurately characterize this update process.

Due to the fact that $(l \mapsto_A m)$ is a linear type, proofs of this type can not be duplicated and may only be consumed by other state axioms. This is the same non-sharing principle that empowers Separation Logic. Unlike Separation Logic, $(l \mapsto_A m)$ itself is a first-class object in CLC, meaning that it can be the result of computation (large eliminations), input to computations, stored in data structures, etc.

4.2 Internalization of Effect Laws

The combination of linear types and dependent types within the state axioms enable the internalization of the so called "effect laws" for state, where these laws are now provable theorems within the CLC framework.

1. Performing get twice on the same address will produce two terms that are provably equal.
2. After using set to update the store at an address l to term m, performing get again on the same address will retrieve m.

Figure 12 is an excerpt taken from our implementation. It demonstrates the internalization of Law 1. The usage of implicit parameters allows one to omit writing much of the redundant arguments that could be inferred from context. Here, new allocates a store for natural number 1, creating a proof c of the form $(l \mapsto_{\mathbb{N}} 1)$. When get is applied to c twice, the equality proofs $pf1 : (1 =_{\mathbb{N}} x)$ and $pf2 : (1 =_{\mathbb{N}} y)$ generated by get allow one to create proof $pf : (x =_{\mathbb{N}} y)$ that the retrieved values x and y are equal. Finally, due to c having linear type, it is required for it to be freed at the very end.

Figure 12. Law 1

```
let [ _, c ] := new _ 1 in
let [ xeq, c ] := get _ _ _ c in
let [ yeq, c ] := get _ _ _ c in
let ( x, pf1 ) := xeq in
let ( y, pf2 ) := yeq in
let pf1 := Eq_sym _ _ _ pf1 in
let pf : Eq _ x y :=
  Eq_trans _ _ _ _ pf1 pf2
in free _ _ _ c.
```

Figure 13 demonstrates the internalization of Law 2. Again, implicit parameters fill in the annoying bureaucratic arguments when they can be inferred. Here, a store is allocated by new for natural number 1, creating a proof c of the form $(l \mapsto_{\mathbb{N}} 1)$. After using set to update the store at l to 2 and then performing get on l, one can create proof $pf : (2 =_{\mathbb{N}} z)$ that the value z retrieved by get is exactly equal to 2.

Though we have not implemented program extraction for CLC yet, our preliminary experiments show the feasibility of

Figure 13. Law 2

```
let [ _, c ] := new _ 1 in
let c := set _ _ _ _ c 2 in
let [ zeq, c ] := get _ _ _ c in
let ( z, pf1 ) := zeq in
let pf : Eq _ 2 z := pf1 in
free _ _ _ c.
```

compiling the state axioms to constant time imperative operations after proof erasure. This can greatly improve the runtime efficiency of CLC over pure functional languages when used for programming. More complex examples such as safe array indexing are available in the first author's Github repository.

4.3 Session Reasoning

The inclusion of dependent types and linear types in CLC makes it extremely expressive when encoding communication protocols. By indexing each channel with a protocol, CLC can enforce communication on a channel to strictly adhere to its specified protocol.

In Figure 14, we declare an inductive `session` type for specifying protocols. Intuitively, the `session` forms a stack of possible operations executed in a protocol. The SEND constructor takes in a non-linear type A and a session ss as its argument, indicating that after sending a message of type A, the protocol progresses to stage ss. Likewise, the RECV constructor takes in a non-linear type A and a session ss as its argument, indicating that after receiving a message of type A, the protocol progresses to stage ss. Finally, the END constructor indicates that the protocol has finished and communication is over.

Figure 14. Inductive Session Type

```
Inductive session : U :=
| SEND : U → session → session
| RECV : U → session → session
| END : session.
```

In Definition 4.2, we postulate five axioms for communication using the session type. These axioms could be viewed as an interface to trusted communication libraries.

Definition 4.2. Communication axioms.

- **channel** : session $\to L$
  The channel axiom is a type constructor. It is used to form the types of communication channels that obey the protocol specified by its session.
- **open** : $(ss : \text{session}) \to \text{channel } ss$
  When given a protocol, open creates a channel indexed by this protocol.
- **close** : channel END $\to$ unit
  Once all communications specified on a given channel has finished, close will close and reclaim the channel.
- **send** : $(A : U) \to A \to (ss : \text{session}) \to$ $\text{channel } (\text{SEND } A\ ss) \to \text{channel } ss$
  After a channel has reached the point in its protocol where a message of type A should be sent, send may send a term of type A on this channel, causing the channel to progress to the next stage of its protocol.
- **recv** : $(A : U) \to (ss : \text{session}) \to$ $\text{channel } (\text{RECV } A\ ss) \to F\_ : A.\text{channel } ss$
  After a channel has reached the point in its protocol where a message of type A should be received, recv may receive such a term of type A from the channel, causing the channel to progress to the next stage of its protocol.

The important detail to notice here is that each channel created by open is of linear type (channel ss) where ss is some arbitrary protocol, hence must be used exactly once. However, the send and recv operations return back channels after consuming them, progressing their protocols one stage forward. Coupled with the fact that the close operation is only able to close channels whose protocols have ended, this forces channels to communicate strictly by their specified protocols.

Dependent types also allows for more precise specifications of protocols themselves. A simple example would be the specification of message sizes. Figure 15 shows such an example. The inductive type `ilist` is a length indexed list. Using `ilist`, we construct a protocol `ss` that first sends two natural numbers, then receives eight natural numbers and finally sends back a boolean acknowledgement.

Figure 15. Sized Communication

```
Inductive ilist (A : U) : Nat → U :=
| nil  : ilist A 0
| cons : A → (n : Nat) →
           ilist A n → ilist A (S n).

let ss :=
  SEND (ilist Nat 2)
     (RECV (ilist Nat 8) (SEND Bool END))
in
let send_msg : ilist Nat 2 :=
  (cons 1 _ (cons 2 _ nil))
in
let ch := open ss in
let ch := send _ send_msg _ ch in
let [ recv_msg, ch ] := recv _ _ ch in
let ch := send _ true _ ch in
close ch
```

4.4 Mutable Data

Immutable data in functional programming languages are easy to reason about due their values staying constant over their lifespan. However, this convenience comes at the price of efficiency when attempting to perform "updates" on existing data. In order to update immutable data, an entirely new copy must be created with new entries substituted in for the original ones. So after update, there are two data structures in memory, the original one and the updated one. A garbage collector is necessary to reclaim memory allocated for the original copy.

Due to the non-sharing property of linear types, it is possible to safely update linear data in-place without encountering synchronization issues. Garbage collection is no longer necessary as extra copies of data are not created. Languages such as ATS that implement mutable linear data have tiny runtimes when compared to immutable garbage collector based ones such as OCaml or Haskell, making them suitable for programming in computationally constrained settings such as embedded programming.

Figure 16. Linear Lists

```
Inductive list (A : U) : L :=
| nil : list A
| cons : A → list A → list A.

Fixpoint append (A : U) : list A -o
  list A -o (list A -o list A) -o list A
:=
  fun ls1 ls2 k ⇒
    match ls1 with
    | nil ⇒ k ls2
    | (cons h t) as _cons_ ⇒
      append _ t ls2
        (fun res ⇒ k (_cons_ h res))
    end.
```

Figure 16 presents a CPS append function for linear lists. In the cons case, the constructor is opened, exposing its data element h and tail list t. Instead of de-allocating the memory of cons, it is bound to the variable _cons_ of linear type $A \multimap list\ A \multimap list\ A$. Basically, each _cons_ must be applied exactly once to reconstruct a list using the memory of the original, thus mutating the original list in-place. Also notice that the continuation k is of linear type ($list\ A \multimap list\ A$), indicating that k must be applied exactly once. Using linear types for continuations is surprisingly accurate as control-flow itself is a linear concept.

5 Related Work

Linear types are a class of type systems inspired by Girard' substructural Linear Logic [11]. Girard notices that the weakening and contraction rules of Classical Logic when restricted carefully, give rise to a new logical foundation for reasoning about resource. Wadler [28, 29] then applies an analogous restriction to variable usage in simple type theory, leading to the development of linear type theory where terms respect resources. A term calculus for linear type theory was later realized by Abramsky [1]. Benton [4] investigates the ramifications of the ! exponential in linear term calculi, decomposing it to adjoint connectives F and G that map between linear and non-linear judgments. Programming languages [5, 19, 31] featuring linear types have also been implemented, allowing programmers to write resource safe software in practical applications.

Over the years, work has been done to enrich linear type theories with dependent types. Cervesato and Pfenning extend the Edinburgh Logical Framework with linear types [7, 13], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár [27] presents a linear dependent type theory, with syntax and semantics drawing inspiration from DILL [3]. Krishnaswami et al. present a dependent linear type theory [14] based on Benton's earlier work on mixed linear and non-linear calculus, demonstrating the ability to internalize imperative programming in the style of Hoare Type Theory [20]. Luo et al. [16] introduce the property of essential linearity and a mixed linear/non-linear context, describing the first type theory that allows types to depend on linear terms. Based on initial ideas of McBride [18], Atkey's Quantitative Type Theory (QTT) [2] uses semi-ring annotations to track variable occurrence, simulating irrelevance, linear and affine types within a unified framework. The Idris 2 programming language [6] implements QTT as its core type system.

6 Future Work

The integration of linear types and dependent types opens the door to many topics for research, both in theory and in its applications.

On the theoretical side, we intend to investigate the semantic models of CLC, with the goal of explaining the natural cohesion between its rules. The impredicative nature of the *Prop*-like universe U_* hints at the possibility of extending dependency to terms of linear types in a philosophically satisfying way. Linear types also appear to augment parametricity, which can produce stronger free theorems.

On the application side, we intend to bolster our implementation with more features, making it a full-featured theorem prover/programming language. We aim to fully verify the typechecking algorithms and extensions employed by the current implementation. The unification algorithm used to resolve implicit parameters is extremely ad-hoc, as the problem of unification for linear types has not thoroughly studied, we hope to improve this situation.

References

[1] Abramsky, S. Computational interpretations of linear logic. *Theoretical Computer Science 111*, 1 (1993), 3–57.

[2] Atkey, R. The syntax and semantics of quantitative type theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom* (2018).

[3] Barber, A. G. Dual intuitionistic linear logic.

[4] Benton, N. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *CSL* (1994).

[5] Bernardy, J., Boespflug, M., Newton, R. R., Jones, S. P., and Spiwack, A. Linear haskell: practical linearity in a higher-order polymorphic language. *CoRR abs/1710.09756* (2017).

[6] Brady, E. C. Idris 2: Quantitative type theory in practice. *CoRR abs/2104.00480* (2021).

[7] Cervesato, I., and Pfenning, F. A linear logical framework. *Information and Computation 179*, 1 (2002), 19–75.

[8] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76*, 2 (1988), 95–120.

[9] de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *Automated Deduction - CADE-25* (Cham, 2015), A. P. Felty and A. Middeldorp, Eds., Springer International Publishing, pp. 378–388.

[10] Dybjer, P. Inductive families. *Formal Aspects of Computing 6* (1997), 440–465.

[11] Girard, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–101.

[12] Girard, J.-Y. Linear logic: its syntax and semantics.

[13] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *J. ACM 40*, 1 (Jan. 1993), 143–184.

[14] Krishnaswami, N. R., Pradic, P., and Benton, N. Integrating linear and dependent types. *SIGPLAN Not. 50*, 1 (Jan. 2015), 17–30.

[15] Luo, Z. An extended calculus of constructions.

[16] Luo, Z., and Zhang, Y. *A Linear Dependent Type Theory*. May 2016, pp. 69–70.

[17] Mazurak, K., Zhao, J., and Zdancewic, S. Lightweight linear types in system fdegree. In *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010* (2010), A. Kennedy and N. Benton, Eds., ACM, pp. 77–88.

[18] McBride, C. I got plenty o' nuttin'. In *A List of Successes That Can Change the World* (2016).

[19] Morrisett, G., Ahmed, A., and Fluet, M. L3: A linear language with locations. In *Typed Lambda Calculi and Applications* (Berlin, Heidelberg, 2005), P. Urzyczyn, Ed., Springer Berlin Heidelberg, pp. 293–307.

[20] Nanevski, A., Morrisett, G., and Birkedal, L. Polymorphism and separation in hoare type theory. *SIGPLAN Not. 41*, 9 (Sept. 2006), 62–73.

[21] Norell, U. Towards a practical programming language based on dependent type theory.

[22] Paulin-Mohring, C. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, B. W. Paleo and D. Delahaye, Eds., vol. 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, Jan. 2015.

[23] Reynolds, J. C. Separation logic: a logic for shared mutable data structures. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), 55–74.

[24] Schäfer, S., Tebbi, T., and Smolka, G. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015* (Aug 2015), X. Zhang and C. Urban, Eds., LNAI, Springer-Verlag.

[25] Takahashi, M. Parallel reductions in λ-calculus. *Inf. Comput. 118*, 1 (Apr. 1995), 120–127.

[26] The Coq Development Team. The Coq Proof Assistant, version 8.11.0.

[27] Vákár, M. Syntax and semantics of linear dependent types. *CoRR abs/1405.0033* (2014).

[28] Wadler, P. Linear types can change the world! In *Programming Concepts and Methods* (1990).

[29] Wadler, P. Is there a use for linear logic? *SIGPLAN Not. 26*, 9 (May 1991), 255–273.

[30] Wadler, P. There's no substitute for linear logic.

[31] Xi, H. Applied type system: An approach to practical programming with theorem-proving. *CoRR abs/1703.08683* (2017).