

The Calculus of Linear Constructions

Qiancheng Fu

October 15, 2021

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.

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 [21] and ECC [14] lie at the core of popular proof assistants such as Coq [24], Agda [20], 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 [11] of purchasing cigarettes literally into CC as a function of type:

$$\textit{Money} \rightarrow \textit{Camels} + \textit{Marlboro}$$

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.

Linear Logic is a substructural logic introduced by Girard in his seminal work [10]. Girard notice that the weakening and contraction rules of Classical Logic when restricted carefully, gives rise to a new logical foundation for reasoning about resource. Wadler [26, 27] first notice that an analogous restriction to variable usage in simple type theory leads to a 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 [18, 29, 5] featuring linear types have also been implemented, allowing programmers to write resource safe software in practical applications. The success of integrating Linear Logic with simple type theory exposes a tantalizing new frontier of integrating linearity with richer type theories.

Work has been done to extend dependent type theories with linear types. Cervesato and Pfenning extend the Edinburgh Logical Framework with linear types [12, 7], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár [25] presents a linear dependent type theory, with syntax and semantics drawing inspiration from DILL [3]. Krishnaswami et al. present a dependent linear type theory [13] 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 [19]. Luo et al. 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 [17], 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.

We propose a new linear dependent type system - The Calculus of Linear Constructions (CLC). CLC extends $CC\omega$ with linear types. $CC\omega$ itself is an extension of CC with a cumulative hierarchy of type universes. We add extra universes L of linear types with cumulativity parallel to the universes 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 $Money : \mathbb{N} \rightarrow L$ as follows.

$$(x :_L Money\ 5) \rightarrow (Camels + Marlboro)$$

This new specification for transaction states that it requires a payment of 5 units of money, the customer is 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 $CC\omega$ with linear types, akin to Mazurak et al.’s work on System F [16]. This allows for a straightforward modeling of CLC in $CC\omega$, and lifting of $CC\omega$ into CLC, 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.

- First, 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.
- Next, we study the meta-theory of CLC directly, showing that it satisfies the standard properties of confluence, propagation, and subject reduction.
- We observe that CLC is highly backwards compatible with $CC\omega$, enabling us to construct a reduction preserving model of CLC in $CC\omega$. This shows that CLC is consistent.
- All major results have been formalized and proven correct in the Coq Proof Assistant with help from the Autosubst [22] 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 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

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 $0, 1, 2, \dots$ that correspond to the predicative universes. We use the meta variable i to quantify over all levels $*, 0, 1, 2, \dots$. Here, U_* is the impredicative universe of propositions, in the same spirit as $CC\omega$'s *Prop* universe. U_k , and L_k are the predicative universes of non-linear, and linear types respectively.

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

Figure 1: Syntax

A clear departure of our language from standard presentations of both linear type theory, and dependent type theory are the presence of two function types: $(x :_s A) \rightarrow B$, $(x :_s A) \multimap B$. 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 theory. Subtle issues arise if $!!$ is not canonically isomorphic to $!$, which may invalidate the substitution lemma [28]. 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 $CC\omega$ without needing any machinery for manipulating exponential.

$$(_ :_U A) \rightarrow B \Leftrightarrow !(A \multimap B) \quad (1)$$

$$(_ :_L A) \rightarrow B \Leftrightarrow !(A \multimap B) \quad (2)$$

$$(_ :_U A) \multimap B \Leftrightarrow !A \multimap B \quad (3)$$

$$(_ :_L A) \multimap B \Leftrightarrow A \multimap B \quad (4)$$

Figure 2: Correspondence of CLC types and MELL implications

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 $*, 0, 1, 2, \dots$. U_* is the impredicative universe of propositions. U_k , and L_k are the predicative universe 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 definitional 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 $A \preceq C$.
2. $U_* \preceq U_0 \preceq U_1 \preceq U_2 \preceq \dots$
3. $L_0 \preceq L_1 \preceq L_2 \preceq \dots$
4. If $A_1 \equiv A_2$ and $B_1 \preceq B_2$, then $(x :_s A_1) \rightarrow B_1 \preceq (x :_s A_2) \rightarrow B_2$
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 universe, and subsequently losing track of its occupants' linearity.

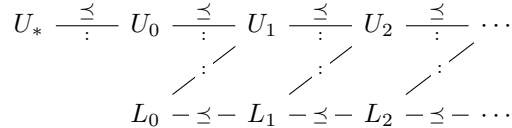


Figure 3: The Universe Hierarchy

2.3 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[15]. 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.

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.

$$\begin{array}{c}
\frac{}{\epsilon \vdash} \text{WF-}\epsilon \qquad \frac{\Gamma \vdash \quad \bar{\Gamma} \vdash A :_{\mathcal{U}} U_i}{\Gamma, x :_{\mathcal{U}} A \vdash} \text{WF-U} \qquad \frac{\Gamma \vdash \quad \bar{\Gamma} \vdash A :_{\mathcal{U}} L_k}{\Gamma, x :_{\mathcal{L}} A \vdash} \text{WF-L} \\
\\
\frac{}{|\epsilon|} \text{PURE-}\epsilon \qquad \frac{|\Gamma| \quad \Gamma \vdash A :_{\mathcal{U}} U}{|\Gamma, x :_{\mathcal{U}} A|} \text{PURE-U} \\
\\
\frac{}{\epsilon \dagger \epsilon \dagger \epsilon} \text{MERGE-}\epsilon \qquad \frac{\Gamma_1 \dagger \Gamma_2 \dagger \Gamma}{\Gamma_1, x :_{\mathcal{U}} A \dagger \Gamma_2, x :_{\mathcal{U}} A \dagger \Gamma, x :_{\mathcal{U}} A} \text{MERGE-U} \qquad \frac{\Gamma_1 \dagger \Gamma_2 \dagger \Gamma \quad x \notin \Gamma_2}{\Gamma_1, x :_{\mathcal{L}} A \dagger \Gamma_2 \dagger \Gamma, x :_{\mathcal{L}} A} \text{MERGE-L1} \\
\\
\frac{\Gamma_1 \dagger \Gamma_2 \dagger \Gamma \quad x \notin \Gamma_1}{\Gamma_1 \dagger \Gamma_2, x :_{\mathcal{L}} A \dagger \Gamma, x :_{\mathcal{L}} A} \text{MERGE-L2}
\end{array}$$

Figure 4: Structural Judgments

Definition 2.2. The context restriction function $\bar{\Gamma}$ 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.

$$\bar{\epsilon} = \epsilon \qquad \overline{\Gamma, x :_{\mathcal{U}} A} = \bar{\Gamma}, x :_{\mathcal{U}} A \qquad \overline{\Gamma, x :_{\mathcal{L}} A} = \bar{\Gamma}$$

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*.

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, 13] for the same reason of avoiding philosophical troubles.

$$\begin{array}{c}
\frac{|\Gamma|}{\Gamma \vdash U_* : U_0} \text{PROP-AXIOM} \quad \frac{|\Gamma|}{\Gamma \vdash s_k : U_{k+1}} \text{SORT-AXIOM} \quad \frac{|\Gamma| \quad \Gamma \vdash A : U_i \quad \Gamma, x :_{\mathcal{U}} A \vdash B : U_*}{\Gamma \vdash (x :_{\mathcal{U}} A) \rightarrow B : U_*} \text{PROP} \\
\\
\frac{|\Gamma| \quad \Gamma \vdash A : U_k \quad \Gamma, x :_{\mathcal{U}} A \vdash B : s_k}{\Gamma \vdash (x :_{\mathcal{U}} A) \rightarrow B : U_k} \text{U-PROD} \quad \frac{|\Gamma| \quad \Gamma \vdash A : L_k \quad \Gamma \vdash B : s_k \quad x \notin \Gamma}{\Gamma \vdash (x :_{\mathcal{L}} A) \rightarrow B : U_k} \text{L-PROD} \\
\\
\frac{|\Gamma| \quad \Gamma \vdash A : U_k \quad \Gamma, x :_{\mathcal{U}} A \vdash B : s_k}{\Gamma \vdash (x :_{\mathcal{U}} A) \multimap B : L_k} \text{U-LOLLI} \quad \frac{|\Gamma| \quad \Gamma \vdash A : L_k \quad \Gamma \vdash B : s_k \quad x \notin \Gamma}{\Gamma \vdash (x :_{\mathcal{L}} A) \multimap B : L_k} \text{L-LOLLI}
\end{array}$$

Figure 5: Type Formation

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 [13], 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 $\Gamma, x :_{\mathcal{U}} A \vdash B : U_*$ asserts that the co-domain must be in the impredicative universe U_* . The final resulting judgment $\Gamma \vdash (x :_{\mathcal{U}} A) \rightarrow B : U_*$ has sort U , indicating that terms with type $(x :_{\mathcal{U}} A) \rightarrow B$ enjoy unrestricted usage.

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 $\Gamma, x :_{\mathcal{U}} A \vdash B : s_k$. 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 $\Gamma \vdash (x :_{\mathcal{U}} A) \rightarrow B : U_k$, 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 $\Gamma \vdash (x :_{\mathcal{L}} A) \rightarrow B : U_k$. 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.

$$\begin{array}{c}
\frac{|\Gamma|}{\Gamma, x :_U A \vdash x : A} \text{U-VAR} \qquad \frac{|\Gamma|}{\Gamma, x :_L A \vdash x : A} \text{L-VAR} \\
\\
\frac{|\Gamma| \quad \Gamma \vdash (x :_s A) \rightarrow B : t_i \quad \Gamma, x :_s A \vdash n : B}{\Gamma \vdash \lambda x.n : (x :_s A) \rightarrow B} \text{PROD-}\lambda \\
\\
\frac{\bar{\Gamma} \vdash (x :_s A) \multimap B : t_i \quad \Gamma, x :_s A \vdash n : B}{\Gamma \vdash \lambda x.n : (x :_s A) \multimap B} \text{LOLLI-}\lambda \\
\\
\frac{\Gamma_1 \vdash m : (x :_U A) \rightarrow B \quad |\Gamma_2| \quad \Gamma_2 \vdash n : A \quad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m n : B[n/x]} \text{U-PROD-APP} \\
\\
\frac{\Gamma_1 \vdash m : (x :_L A) \rightarrow B \quad \Gamma_2 \vdash n : A \quad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m n : B[n/x]} \text{L-PROD-APP} \\
\\
\frac{\Gamma_1 \vdash m : (x :_U A) \multimap B \quad |\Gamma_2| \quad \Gamma_2 \vdash n : A \quad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m n : B[n/x]} \text{U-LOLLI-APP} \\
\\
\frac{\Gamma_1 \vdash m : (x :_L A) \multimap B \quad \Gamma_2 \vdash n : A \quad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m n : B[n/x]} \text{L-LOLLI-APP} \\
\\
\frac{\Gamma \vdash m : A \quad \bar{\Gamma} \vdash B : s_i \quad A \preceq B}{\Gamma \vdash m : B} \text{CONV}
\end{array}$$

Figure 6: Term Formation

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 $\Gamma, x :_U A$ with a non-linear type A . The L-VAR rule asserts that free variable x occurs within the context $\Gamma, x :_L A$ with linear type A . For both rules, the side condition $|\Gamma|$ forbids irrelevant variables of linear types from occurring within Γ . This prevents another vector of weakening variables with linear type.

In PROD- λ , the function type being addressed has form $(x :_s A) \rightarrow B$. 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 LOLL- 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 $\bar{\Gamma} \vdash (x :_s A) \multimap B : L_k$ 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 $\bar{\Gamma}$. Furthermore, 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.

$$\begin{array}{c}
\frac{m_1 \rightsquigarrow^* n \quad m_2 \rightsquigarrow^* n}{m_1 \equiv m_2 : A} \text{JOIN} \qquad \frac{}{(\lambda x.m) n \rightsquigarrow m[n/x]} \text{P-}\beta \qquad \frac{m \rightsquigarrow m'}{\lambda x.m \rightsquigarrow \lambda x.m'} \text{STEP-}\lambda \\
\\
\frac{A \rightsquigarrow A'}{(x :_s A) \rightarrow B \rightsquigarrow (x :_s A') \rightarrow B} \text{STEP-PROD}L \qquad \frac{B \rightsquigarrow B'}{(x :_s A) \rightarrow B \rightsquigarrow (x :_s A) \rightarrow B'} \text{STEP-PROD}R \\
\\
\frac{A \rightsquigarrow A'}{(x :_s A) \multimap B \rightsquigarrow (x :_s A') \multimap B} \text{STEP-LOLLI}L \qquad \frac{B \rightsquigarrow B'}{(x :_s A) \multimap B \rightsquigarrow (x :_s A) \multimap B'} \text{STEP-LOLLI}R \\
\\
\frac{m \rightsquigarrow m'}{m n \rightsquigarrow m' n} \text{STEP-APPL} \qquad \frac{n \rightsquigarrow n'}{m n \rightsquigarrow m n'} \text{STEP-APPR}
\end{array}$$

Figure 7: Equality and Parallel Reduction

As we have discussed previously, the elimination of the explicit $!$ exponential allows CLC to maintain the standard operational semantics of $\text{CC}\omega$, whose β -reductions are very well behaved.

2.8 Meta Theory

In this section, we focus our discussion on the meta-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 *derelection* rules can be encoded as η -expansions. Finally, we construct a reduction preserving erasure function that maps well-typed CLC terms to well-typed $\text{CC}\omega$ terms, showing that CLC is strongly normalizing.

All proofs have been formalized in Coq with help from the Autosubst [22] 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 given 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 [23]. The presence of linear types do not pose any complications as reductions are untyped.

Lemma 2.1. *Parallel reduction satisfies the diamond property.*

If $m \rightsquigarrow_p m_1$ and $m \rightsquigarrow_p m_2$ then there exists m' such that $m_1 \rightsquigarrow_p m'$ and $m_2 \rightsquigarrow_p m'$.

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 $\Gamma, x :_{\mathcal{U}} B \vdash m : A$ is derivable.

2.8.3 Substitution

Though the substitution lemma is widely considered a boring and bureaucratic theorem, it is surprisingly hard to design linear typed languages where the substitution lemma is admissible. Much of this difficulty arise during the substitution of arguments containing ! exponential. Perhaps the most famous work detailing the issues of substitution is due to Wadler [28]. He defines additional syntax and semantics for the intricate unboxing of ! terms, solving the lack of substitute in Abramsky's term calculus [1].

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 $\Gamma_1, x :_{\mathcal{U}} A \vdash m : B$ 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 $\Gamma_1, x :_{\mathcal{L}} A \vdash m : B$ 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 exists some sort s and level i such that $\bar{\Gamma} \vdash A : s_i$.

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 yield 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 $occurs(x, m) = 1 \wedge occurs(x, n) = 0$ or $occurs(x, m) = 0 \wedge occurs(x, n) = 1$ 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

In most presentations of linear logic, non-linear objects are created by packing linear objects into the $!$ exponential using the *promotion* rule. Conversely, the *dereliction* rule unpacks non-linear objects so they can be used in place of linear ones. Linear type theories [1, 3, 25] utilizing exponential for non-linearity will contain $!$ types and rules that are equivalent to *promotion* and *dereliction*.

Linear types of CLC are not designed around packing and unpacking $!$, so there are no rules for *promotion* and *dereliction*. 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 $\Gamma \vdash n : (x :_s A) \rightarrow B$ is derivable.

Theorem 2.12. *Dereliction*

If $\Gamma \vdash m : (x :_s A) \rightarrow B$ 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 $CC\omega$. We assume familiarity with $CC\omega$ syntax here, and define the erasure function as follows.

Definition 2.4.

$$\begin{aligned}
\llbracket x \rrbracket &= x \\
\llbracket U_* \rrbracket &= Prop \\
\llbracket U_k \rrbracket &= Type_k \\
\llbracket L_k \rrbracket &= Type_k \\
\llbracket (x :_s A) \rightarrow B \rrbracket &= (x : \llbracket A \rrbracket) \rightarrow \llbracket B \rrbracket \\
\llbracket (x :_s A) \multimap B \rrbracket &= (x : \llbracket A \rrbracket) \rightarrow \llbracket B \rrbracket \\
\llbracket \lambda x. n \rrbracket &= \lambda x. \llbracket n \rrbracket \\
\llbracket m \ n \rrbracket &= \llbracket m \rrbracket \llbracket n \rrbracket
\end{aligned}$$

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

Definition 2.5.

$$\begin{aligned} \llbracket \epsilon \rrbracket &= \epsilon \\ \llbracket \Gamma, x :_s A \rrbracket &= \llbracket \Gamma \rrbracket, x : \llbracket A \rrbracket \end{aligned}$$

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 , $\llbracket m[n/x] \rrbracket = \llbracket m \rrbracket [\llbracket n \rrbracket / x]$.

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

Lemma 2.14. *Erasure preserves reduction.*

For any CLC terms m , and n , if there is $m \rightsquigarrow_{\text{CLC}} n$, then there is $\llbracket m \rrbracket \rightsquigarrow_{\text{CC}\omega} \llbracket n \rrbracket$.

Lemma 2.15. *Embedding.*

If $\Gamma \vdash m :_s A$ is a valid judgment in CLC, then $\llbracket \Gamma \rrbracket \vdash \llbracket m \rrbracket : \llbracket A \rrbracket$ is a valid judgment in $\text{CC}\omega$.

If there exists some well typed CLC term with an infinite sequence of reductions, erasure will embed this term into a well typed $\text{CC}\omega$ term along with its infinite sequence of reductions by virtue of lemmas 2.14, and 2.15. This is contradictory to the strong normalization property of $\text{CC}\omega$ proven through the Girard-Tait method [14], so this hypothetical term does not exist in CLC.

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

3 Future Work

4 Conclusion

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] GIRARD, J.-Y. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.
- [11] GIRARD, J.-Y. Linear logic: its syntax and semantics.
- [12] HARPER, R., HONSELL, F., AND PLOTKIN, G. A framework for defining logics. *J. ACM* 40, 1 (Jan. 1993), 143–184.
- [13] KRISHNASWAMI, N. R., PRADIC, P., AND BENTON, N. Integrating linear and dependent types. *SIGPLAN Not.* 50, 1 (Jan. 2015), 17–30.
- [14] LUO, Z. An extended calculus of constructions.
- [15] LUO, Z., AND ZHANG, Y. *A Linear Dependent Type Theory*. May 2016, pp. 69–70.
- [16] 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.
- [17] MCBRIDE, C. I got plenty o’ nuttin’. In *A List of Successes That Can Change the World* (2016).
- [18] 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.
- [19] NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. Polymorphism and separation in hoare type theory. *SIGPLAN Not.* 41, 9 (Sept. 2006), 62–73.
- [20] NORELL, U. Towards a practical programming language based on dependent type theory.
- [21] 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.
- [22] 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.
- [23] TAKAHASHI, M. Parallel reductions in λ -calculus. *Inf. Comput.* 118, 1 (Apr. 1995), 120–127.
- [24] THE COQ DEVELOPMENT TEAM. The Coq Proof Assistant, version 8.11.0.
- [25] VÁKÁR, M. Syntax and semantics of linear dependent types. *CoRR abs/1405.0033* (2014).
- [26] WADLER, P. Linear types can change the world! In *Programming Concepts and Methods* (1990).
- [27] WADLER, P. Is there a use for linear logic? *SIGPLAN Not.* 26, 9 (May 1991), 255–273.
- [28] WADLER, P. There’s no substitute for linear logic.
- [29] XI, H. Applied type system: An approach to practical programming with theorem-proving. *CoRR abs/1703.08683* (2017).