The Calculus of Linear Constructions

Qiancheng Fu

October 12, 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, and an indexed typing judgment 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 and show that CLC as a programming language enables the manipulation of mutable data structures in a principled way.

1 Introduction

The Calculus of Constructions (CC) is a dependent type theory introduced by Coquand, and Huet in their landmark work [10]. In CC types can depend on terms, allowing one to write precise specifications as types. Today, CC and its variations CIC [25] and ECC [18] lie at the core of popular proof assistants such as Coq [28], Agda [24], Lean [11], and others. These theorem provers have found great success in the fields of software verification [17, 2], and constructive mathematics [14, 7].

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 [13] of purchasing cigarettes literally into CC as a function of type:

$$Money \to Camels + Marlboro$$

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 [9] to provide additional reasoning principles for dealing with resource. The design and embedding of these logics is a difficult problem in its own right, requiring additional proofs to justify its soundness. We propose an alternative solution: extend CC with linear types.

Linear Logic is a substructural logic introduced by Girard in his seminal work [12]. 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 [30, 31] 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 [22, 33, 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 have been done to extend dependent type theories with linear types. Cervesato and Pfenning extends the Edinburgh Logical Framework with linear types [15, 8], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár [29] gives a categorical semantics for linear dependent types. Krishnaswami et al. present a dependent linear type theory [16] based on Benton's early work of mixed linear and non-linear calculus, demonstrating the ability to internalize imperative programming the style of Hoare Type Theory [23]. 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 [21], Atkey's Quantitative Type Theory (QTT) [3] 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 CCω with linear types. CCω 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. Universe information is propagated by an indexed typing judgment down to the term level, controlling the usage of weakening and contraction rules. This ultimately results 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 encodes the usage of resource. The previous example of monetary transaction can be refined using an indexed linear type family $Money : \mathbb{N} \to L$ as follows.

$$(Money\ 5) \multimap (Camels + Marlboro)$$

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 CCω with linear types, akin to Mazurak et al.'s work on System F [20]. This allows for a straightforward modeling of CLC in CCω, and lifting of CCω into CLC, endowing CLC with the fruits of decades of CC research. We further extend CLC with inductive types, showing that as a programming language it can manipulate mutable data structures in a principled way. 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.
- Next, we study the meta-theory of CLC directly, showing that it satisfies the standard properties of confluence, regularity, and subject reduction.
- We observe that CLC is highly backwards compatible with CCω. We construct a reduction preserving model of CLC in CCω, showing that CLC is consistent.
- All major results have been formalized and proven correct in the Coq Proof Assistant with help from the Autosubst [26] 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 that as a programming language, CLC can safely manipulate mutable data structures.
- 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 $0, 1, 2, ...$ that correspond to the predicative universes. We use the meta variable i to quantify over all levels $*, 0, 1, 2, ...$ 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.

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

Figure 1: Syntax

A clear departure of our language from standard presentations of both linear type theory, and dependent type theory is the indexed function type: $(x : A)\, {}_s\!\!\rightarrow_t B$. The reason for these indices is that we have built the ! exponential of linear logic directly into universe sorts. The indices s, and t annotate the domain, and co-domain's respective universe sort, hence annotating linearity. 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 [32]. By integrating the exponential directly into universe sorts, we implicitly limit ! to only be used canonically. This allows us to derive the substitution lemma, and construct a direct modeling of CLC in CCω without needing any machinery for manipulating exponential.

$$
\begin{array}{rcll}
(\_ : A)\, {}_U\!\!\rightarrow_U B & \Leftrightarrow & !(!A \multimap !B) & (1)\\
(\_ : A)\, {}_U\!\!\rightarrow_L B & \Leftrightarrow & !(!A \multimap B) & (2)\\
(\_ : A)\, {}_L\!\!\rightarrow_U B & \Leftrightarrow & !(A \multimap !B) & (3)\\
(\_ : A)\, {}_L\!\!\rightarrow_L B & \Leftrightarrow & !(A \multimap B) & (4)\\
(\_ : A)\, {}_U\!\!\multimap_U B & \Leftrightarrow & !A \multimap !B & (5)\\
(\_ : A)\, {}_U\!\!\multimap_L B & \Leftrightarrow & !A \multimap B & (6)\\
(\_ : A)\, {}_L\!\!\multimap_U B & \Leftrightarrow & A \multimap !B & (7)\\
(\_ : A)\, {}_L\!\!\multimap_L B & \Leftrightarrow & A \multimap B & (8)
\end{array}
$$

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), (2) if the co-domain B is dependent on arguments of domain A. Though it may seem as if implications of the $!(A \multimap B)$ form are left out, CLC encodes these by endowing each function type with a universe sort through

their formation judgment. The encoding for this example would be $\Gamma \vdash A \,_L\!\to_L B :_U U_i$. We will discuss function type formation in Section 2.5 in greater detail.

In practice, algorithmic type checking techniques such as bi-directional typing allow users to omit writing most of these sort indices.

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 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 $A \preceq B$.
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 $(x : A_1) \,_s\!\to_t B_1 \preceq (x : A_2) \,_s\!\to_t B_2$

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.

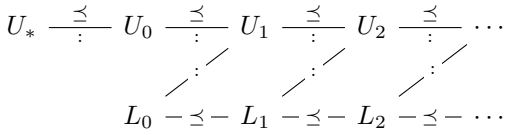

Figure 3: The Universe Hierarchy

2.3 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[19]. 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.

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

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

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

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

Figure 4: Structural Judgments

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 $\Gamma \vdash m :_s A$. Intuitively, this judgment states that the term m is an inhabitant of type A, with free variables typed in Γ. The sort index s tells us the linearity of m. Specifically, if $s = U$, then m has unrestricted usage. Conversely, if $s = L$, then m must be used exactly once. Theorem 2.4 (regularity) shows that s corresponds exactly to the sort of A's universe.

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 $\Gamma \vdash A :_U U_i$.
2. A *type* A is *linear* under context Γ if $\Gamma \vdash A :_U L_k$.
3. A *term* m is *unrestricted* under context Γ if it has type A, and $\Gamma \vdash m :_U A$. A unrestricted term may be used an arbitrary number of times.
4. A *term* m is *restricted* under context Γ if it has type A, and $\Gamma \vdash m :_L A$ 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 [8, 16] for the same reason of avoiding philosophical troubles.

$$\frac{|\Gamma|}{\Gamma \vdash U_* :_U U_0}\textsc{Prop-Axiom} \qquad \frac{|\Gamma|}{\Gamma \vdash U_k :_U U_{k+1}}\textsc{U-Axiom} \qquad \frac{|\Gamma|}{\Gamma \vdash L_k :_U U_{k+1}}\textsc{L-Axiom}$$

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

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

$$\frac{|\Gamma| \qquad \Gamma \vdash A :_U L_k \qquad \Gamma \vdash B :_U s_k \qquad x \notin \Gamma}{\Gamma \vdash (x : A)\, L \to_s B :_U t_k}\textsc{L-Prod}$$

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 reminiscent of Krishnaswami et al.'s treatment of linear universes [16], where linear *types* themselves can be used unrestrictedly.

The U-Prop rule is used for forming propositions. From the judgment $\Gamma \vdash A :_U U_i$ we can see that U-Prop allows for quantification over non-linear types of arbitrary level, hence impredicative. The judgment $\Gamma, x :_U A \vdash B :_U U_*$ asserts that the co-domain must be in the impredicative universe U_*. The final resulting judgment $\Gamma \vdash (x : A)\, U \to_U B :_U U_*$ is indexed by U, indicating that the *type* $(x : A)\, U \to_U B$ can be use unrestrictedly as a *term*. Since $(x : A)\, U \to_U B$ belongs to universe U_*, its *terms* also enjoys unrestricted usage.

The U-Prod rule is used for forming function types with non-linear domains. This is evident from the judgment $\Gamma \vdash A :_U U_k$. Since 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 $\Gamma, x :_U A \vdash B :_U s_k$. 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 $\Gamma \vdash (x : A)\, U \to_s B :_U t_k$, we see that A, and B's universe sorts are used as indices to the arrow. Interestingly, $(x : A)\, U \to_s B$ is assigned to a universe t_k of level k and arbitrary sort t. If t is chosen to be U, then $\Gamma \vdash (x : A)\, U \to_s B :_U U_k$ tells us λ-abstractions of this type have unrestricted usage. Conversely, if t is chose to be L, then $\Gamma \vdash (x : A)\, U \to_s B :_U L_k$ tells us λ-abstractions of this type have restricted usage. This is how we encode the missing $!(A \multimap B)$ forms from MELL shown in Figure 2.

The L-Prod rule is used for forming function types with linear domains. From the judgment $\Gamma \vdash A :_U L_k$, 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 $\Gamma \vdash B :_U s_k$. 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 $\Gamma \vdash (x : A)\, L \to_s B :_U t_k$. Here, x is a hypocritical unbinding variable whose only purpose is to preserve syntax uniformity. Again, we may chose t to be either U or L, which form function types for λ-abstractions with unrestricted or restricted usage respectively.

For rules U-Prod, and L-Prod, it may seem as if the final choice of t_k does not matter. Indeed, the choice of $t = U$ or $t = L$ has no effect on type formation. But the differences will become apparent when typing λ-abstractions in Section 2.6.

2.6 Term Formation

The rules for term formation are presented in Figure 6.

$$\frac{|\Gamma|}{\Gamma, x :_U A \vdash x :_U A}\text{U-Var} \qquad \frac{|\Gamma|}{\Gamma, x :_L A \vdash x :_L A}\text{L-Var}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash (x : A)\, s \to t\, B :_U U_i \qquad \Gamma, x :_s A \vdash n :_t B}{\Gamma \vdash \lambda x.n :_U (x : A)\, s \to t\, B}\text{U-}\lambda$$

$$\frac{\overline{\Gamma} \vdash (x : A)\, s \to t\, B :_U L_k \qquad \Gamma, x :_s A \vdash n :_t B}{\Gamma \vdash \lambda x.n :_L (x : A)\, s \to t\, B}\text{L-}\lambda$$

$$\frac{\Gamma_1 \vdash m :_t (x : A)\, U \to s\, B \qquad |\Gamma_2| \qquad \Gamma_2 \vdash n :_U A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B[n/x]}\text{U-App}$$

$$\frac{\Gamma_1 \vdash m :_t (x : A)\, L \to s\, B \qquad \Gamma_2 \vdash n :_L A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B[n/x]}\text{L-App}$$

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

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 with linear type from occurring within Γ. This prevents weakening variables with linear type.

As promised in Section 2.5, the choice of t_k for function type formation comes into play during the typing of λ-abstractions. Namely, in the pair of rules U-λ, and L-λ.

By design, types with universe sort U are non-linear, terms with these types enjoy unrestricted usage. In U-λ, the function type being addressed has universe U_i, as seen by the judgment $\Gamma \vdash (x : A)\, s \to t\ :_U U_i$. λ-abstractions of this type can be applied an unrestricted number of times, hence cannot depend on free variables with restricted usage. 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 $\Gamma, x :_s A \vdash n :_t B$, where s is the sort of A and t is the sort of B. Finally, the resulting judgment $\Gamma \vdash \lambda x.n :_U (x : A)\, s \to t\, B$ with index U, asserts that the λ-abstraction can be used unrestrictedly.

In contrast to U-λ, the L-λ rule is used for forming λ-abstractions that must be used exactly once. Since these abstractions must be used once, they are allowed access to restricted variables within context Γ, evident in the judgment $\Gamma, x :_s A \vdash n :_t B$, and lack of side condition $|\Gamma|$. However, in judgment $\overline{\Gamma} \vdash (x : A)\, s \to t\, B :_U L_k$ the context must be filtered, since types are not allowed to depend on restricted variables. The final resulting judgment $\Gamma \vdash \lambda x.n :_L (x : A)\, s \to t\, B$ with index L, asserts that the λ-abstraction must be used exactly once.

For U-App, domain A is a non-linear type, as seen by its arrow index in $(x : A)\, U \to s\, B$. 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-App, domain A is a linear type, as seen by its arrow index in $(x : A) \, {}_L\!\!\rightarrow_s B$. 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.

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

2.7 Equality and Parallel Reduction

The operational semantics, and definitional equality of CLC is defined by parallel reductions [27] as presented in Figure 7, all of which are entirely standard.

$$\frac{m_1 \rightsquigarrow^* n \qquad m_2 \rightsquigarrow^* n}{m_1 \equiv m_2 : A}\textsc{Join} \qquad \frac{}{x \rightsquigarrow x}\textsc{P-Var} \qquad \frac{}{U \rightsquigarrow U}\textsc{P-U} \qquad \frac{}{L \rightsquigarrow L}\textsc{P-L} \qquad \frac{n \rightsquigarrow n'}{\lambda x.n \rightsquigarrow \lambda x.n'}\textsc{P-}\lambda$$

$$\frac{m \rightsquigarrow m' \qquad n \rightsquigarrow n'}{m\ n \rightsquigarrow m'\ n'}\textsc{P-App} \qquad \frac{m \rightsquigarrow m' \qquad n \rightsquigarrow n'}{(\lambda x.m)\ n \rightsquigarrow m'[n'/x]}\textsc{P-}\beta \qquad \frac{A \rightsquigarrow A' \qquad B \rightsquigarrow B'}{(x : A)\ {}_s\!\!\rightarrow_t B \rightsquigarrow (x : A')\ {}_s\!\!\rightarrow_t B'}\textsc{P-Prod}$$

Figure 7: Equality and Parallel Reduction

As we have discussed previously, the elimination of the explicit ! exponential allows CLC to maintain a simple operational semantics, 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. 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 [26] 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. The presence of linear types do not pose any complications as reductions are untyped.

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

Corollary 2.1.1. *The transitive reflexive closure of parallel 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 2.1.2. *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 2.2. *Weakening.*
If $\Gamma \vdash m :_s A$ is a valid, then for any $x \notin \Gamma$, judgment $\Gamma, x :_U B \vdash m :_s A$ 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 [32]. 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 substitution lemma is directly proved by induction on typing derivations.

Lemma 2.3. *Substitution.*
For $\Gamma_1, x :_s A \vdash m :_t B$ and $\Gamma_2 \vdash v :_s A$, if $\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma$ is defined for some Γ, then $\Gamma \vdash [v/x]m :_t [v/x]B$.

2.8.4 Type Soundness

In order to prove subject reduction, we first prove the regularity theorem. The main purpose of regularity is to lower types down to the term level, enabling the application of various inversion lemmas. Regularity also serves a second purpose, it tells us the judgment index s of a term $\Gamma \vdash m :_s A$ is exactly the same sort as its type's universe $\overline{\Gamma} \vdash A :_U s_i$.

Theorem 2.4. *Regularity.*
For any context Γ, term m, type A, and sort s, if $\Gamma \vdash m :_s A$ is a valid judgment, then there exists some level i such that $\overline{\Gamma} \vdash A :_U s_i$.

With weakening, substitution, regularity, and various inversion lemmas proven, subject reduction is proved by induction on typing derivation.

Theorem 2.5. *Subject reduction.*
For $\Gamma \vdash m :_s A$, if $m \rightsquigarrow n$ then $\Gamma \vdash n :_s A$.

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 2.6. *Narity.*
If $\Gamma \vdash m :_s A$ 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 2.7. *Linearity.*
If $\Gamma \vdash m :_s A$ *is a valid judgment, for any* $(x :_L B) \in \Gamma$*, there is* $occurs(x, m) = 1$*.*

2.8.6 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.

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

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

Definition 2.5.

$$\begin{aligned}
[\![\epsilon]\!] &= \epsilon \\
[\![\Gamma, x :_s A]\!] &= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}$$

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

Lemma 2.8. *Erasure commutes with substitution.*
For any CLC terms m*,* n*, and some variable* x*,* $[\![m[n/x]]\!] = [\![m]\!][[\![n]\!]/x]$*.*

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

Theorem 2.9. *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 2.10. *Embedding.*
For any CLC context Γ*, terms* m*,* A*, and sort* s*, if* $\Gamma \vdash m :_s A$ *is a valid typing judgment in CLC, then* $[\![\Gamma]\!] \vdash [\![m]\!] : [\![A]\!]$ *is a valid a typing judgment in CC*ω*.*

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 2.9, and 2.10. This is contradictory to the strong normalization property of CCω proven through the Girard-Tait method [18], so this hypothetical term does not exist in CLC.

Theorem 2.11. *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] Appel, A., Beringer, L., Chlipala, A., Pierce, B., Shao, Z., Weirich, S., and Zdancewic, S. Position paper: the science of deep specification. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences 375* (10 2017), 20160331.

[3] 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).

[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] Buzzard, K., Hughes, C., Lau, K., Livingston, A., Mir, R. F., and Morrison, S. Schemes in lean, 2021.

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

[9] Chlipala, A. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not. 46*, 6 (June 2011), 234–245.

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

[11] 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.

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

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

[14] Gonthier, G. A computer-checked proof of the four colour theorem.

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

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

[17] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress* (Toulouse, France, Jan. 2016), SEE.

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

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

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

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

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

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

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

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