The Calculus of Linear Constructions

Anonymous Author(s)

Abstract

The Calculus of Linear Constructions (CLC) is an extension of the Calculus of Constructions (CC) with linear types. Specifically, CLC extends the predicative CCω 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. We further extend CLC with rules for defining inductive linear types in the style of CIC, forming the Calculus of Inductive Linear Constructions (CILC). Through examples, we demonstrate that CILC facilitates correct by construction imperative programming and lightweight protocol enforcement. We have formalized and proven correct all major results in the Coq Proof Assistant.

Keywords: Dependent types, linear types, inductive types, Calculus of Constructions

ACM Reference Format:
Anonymous Author(s). 2022. The Calculus of Linear Constructions. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 23 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 extensions CIC [19] and ECC [13] lie at the core of popular proof assistants such as Coq [23], Agda [18], 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 can be inconvenient for CC to encode and reason about resource. Users of proof assistants based on CC often need to embed external logics such as Separation Logic to provide domain-specific reasoning principles for dealing with resource. We propose an alternative solution: extend CC with linear types.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper presents a new linear dependent type system — the Calculus of Linear Constructions (CLC). CLC extends the predicative 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 every resource is used exactly once.

Preexisting approaches to integrating linear types and dependent types often relied on the ! exponential or separate typing judgments for linear and non-linear types. The universe sorts of CLC are enough to completely distinguish between linear and non-linear types within a single typing judgment. As a result, CLC falls much closer to its CC roots than prior works in both syntax and semantics.

We further extend CLC with rules for defining inductive linear types in the style of CIC, forming the Calculus of Inductive Linear Constructions (CILC). Standard Linear Logic connectives such as $\otimes$ and $\oplus$ are definable using this generalized mechanism in a straightforward way. Connectives that mix linear and non-linear types can be constructed just as easily using CILC.

Through examples in Section 4, we show some applications of CILC such as construction of safe random access array from the first principles, protocol enforcement through inductively defined session types and in-place update of data structures using continuations.

We have formalized all major results in Coq and implemented a prototype in OCaml. Additional examples of applications such as provable effect laws and quantitative reasoning are included with our implementation.

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, enabling direct and precise reasoning about resource.
- Second, we study the meta-theory of CLC, showing that it exhibits many desirable properties such as confluence, subject reduction and logical consistency.
- Furthermore, we extend CLC with rules to define inductive linear types, forming the Calculus of Inductive Linear Constructions (CILC). This is the first complete presentation of a generalized mechanism for defining inductive linear types.
- All major results have been proven correct in the Coq. Our development is the first machine checked formalization of a linear dependent type theory. We have also implemented a prototype of CILC in OCaml.

2 The Language of CLC

2.1 Syntax

Figure 1. Syntax of CLC

i	::=	$0 \mid 1 \mid 2 \ldots$	universe levels
s, t	::=	$U \mid L$	sorts
m, n, A, B, M	::=	$U_i \mid L_i \mid x$	expressions
		$\mid (x :_s A) \to B$	
		$\mid (x :_s A) \multimap B$	
		$\mid \lambda x :_s A.n$	
		$\mid m\ n$	

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 i to quantify over universe levels $0, 1, 2, \ldots$. U_i and L_i are the universes at level i of non-linear and linear types respectively.

Figure 2. Correspondence of CLC types and MELL implications

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

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: $(x :_s A) \to B$ and $(x :_s A) \multimap B$. The reason for these function types is that we have built the ! exponential of linear logic implicitly 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 [27]. By integrating the exponential directly into universe sorts, we have limited ! 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 functions 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.

Algorithmic type checking techniques allow users to omit writing sort indices for many situations in practice. Our implementation employs bi-directional type checking and users rarely interact with sort 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_i and L_i 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 universes 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_i are restricted to being used exactly once.

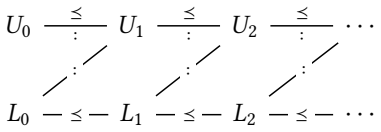
In order to lift terms from lower universes to higher ones, there exists cumulativity between universe levels of the same sort. We define cumulativity as follows.

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. $U_0 \preceq U_1 \preceq U_2 \preceq \cdots$
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_i has U_{i+1} as its type, allowing functions to freely quantify over linear *types*. However, L_i cumulates to L_{i+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



2.3 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[14]. 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 *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ 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 *merge* relation is defined if and only if each variable occurs uniquely in one context and not the other. This definition of *merge* 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.

Figure 4. Structural Judgments

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

$$\frac{\Gamma \vdash \qquad \overline{\Gamma} \vdash A : L_i}{\Gamma, x :_L A \vdash}\textsc{Wf-L} \qquad \frac{}{|\epsilon|}\textsc{Pure-}\epsilon$$

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

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

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

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

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 terminology *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 $\Gamma \vdash m : A$. An unrestricted term may be used any 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 similarly to [7, 12] for the same reason of avoiding philosophical troubles.

The axiom rules U-Axiom and L-Axiom are similar to standard CCω, 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 weakening rule. 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 U-Axiom and L-Axiom. From L-Axiom we can see that the universe of linear types L_i is an inhabitant of U_{i+1}. This is inspired by Krishnaswami et al.'s treatment of linear universes [12], where linear *types* themselves can be used unrestrictedly.

Figure 5. Type Formation

$$\frac{|\Gamma|}{\Gamma \vdash s_i : U_{i+1}}\textsc{Sort-Axiom}$$

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

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_i \qquad \Gamma \vdash B : s_i \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \to B : U_i}\textsc{L}\to$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma, x :_U A \vdash B : s_i}{\Gamma \vdash (x :_U A) \multimap B : L_i}\textsc{U}\multimap$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_i \qquad \Gamma \vdash B : s_i \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \multimap B : L_i}\textsc{L}\multimap$$

The U$\to$ rule is used for forming non-linear function types with non-linear domains. This is evident from the judgment $\Gamma \vdash A : U_i$. Due to the fact that A is in the non-linear universe U_i, 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 $\Gamma, x :_U A \vdash B : s_i$. The domain B itself may be linear of non-linear, since B's universe s_k can vary between U_i and L_i. From the resulting judgment $\Gamma \vdash (x :_U A) \to B : U_i$, 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→ rule is used for forming non-linear function types with linear domains. From the judgment $\Gamma \vdash A : L_i$, 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, evidently in the judgment $\Gamma \vdash B : s_i$. Like U→, co-domain B itself may be linear or non-linear, since B's universe s_i can vary between U_i and L_i. The final resulting judgment is $\Gamma \vdash (x :_L A) \to B : U_i$. 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⊸, L⊸ rules are similar to U→, L→ in spirit. The dependency considerations for domain A and co-domain B are exactly the same. The main difference between U⊸, L⊸ and U→, L→ is the universe sort of the resulting type. The sorts of the function types formed by U⊸, L⊸ 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.

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 $\lambda\to$, the function type has the form $(x :_s A) \to B$. 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 $\Gamma \vdash \lambda x.n : (x :_s A) \to B$ asserts that the λ-abstraction can be used unrestrictedly.

In contrast to $\lambda\to$, the $\lambda\multimap$ rule is used for forming abstractions that must be used exactly once. Due to the fact that these abstractions must be used once, they are allowed access to restricted variables within context Γ, evidently 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 App-U→, domain A is a non-linear type, as seen by its annotation in $(x :_U A) \to B$. 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 $merge\ \Gamma_1\ \Gamma_2\ \Gamma$, contracting all unrestricted variables shared between Γ_1 and Γ_2.

Now for App-L→, domain A is a linear type, as seen by its annotation in $(x :_L A) \to B$. 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 $merge\ \Gamma_1\ \Gamma_2\ \Gamma$, contracting all unrestricted variables shared between Γ_1 and Γ_2.

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

Finally, the Conversion rule allows judgment $\Gamma \vdash m : A$ to convert to judgment $\Gamma \vdash m : B$ if B is a valid type in context $\overline{\Gamma}$.

Figure 6. Term Formation

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

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

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

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

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

$$\frac{\Gamma_1 \vdash m : (x :_U A) \multimap B \qquad |\Gamma_2| \qquad \Gamma_2 \vdash n : A \qquad merge\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n : B[n/x]}\text{App-U}\multimap$$

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

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

Futhermore, A must be a subtype of B satisfying the relation $A \preceq B$. This rule gives rise to large eliminations (compute types from terms), as computations embedded at the type level can reduce to canonical types.

2.7 Equality and Reduction

The operational semantics and β-equality of CLC terms 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 Reduction

$$\frac{m_1 \rightsquigarrow^* n \qquad m_2 \rightsquigarrow^* n}{m_1 \equiv m_2 : A}\textsc{Join}$$

$$\frac{}{(\lambda x :_s A.m)\ n \rightsquigarrow m[n/x]}\textsc{Step-}\beta$$

$$\frac{A \rightsquigarrow A'}{\lambda x :_s A.m \rightsquigarrow \lambda x :_s A'.m}\textsc{Step-}\lambda\textsc{L}$$

$$\frac{m \rightsquigarrow m'}{\lambda x :_s A.m \rightsquigarrow \lambda x :_s A.m'}\textsc{Step-}\lambda\textsc{R}$$

$$\frac{A \rightsquigarrow A'}{(x :_s A) \to B \rightsquigarrow (x :_s A') \to B}\textsc{Step-L}\to$$

$$\frac{B \rightsquigarrow B'}{(x :_s A) \to B \rightsquigarrow (x :_s A) \to B'}\textsc{Step-R}\to$$

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

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

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

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 an effective 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 [21] 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 an accompanying technical report.

2.8.1 Reduction and Confluence.

The proof of confluence is entirely standard using parallel step technique [22]. The presence of linear types does not pose any complications as reductions are untyped.

Theorem 2.4. *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 $m_2 \rightsquigarrow^* m'$.*

2.8.2 Weakening.

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

Theorem 2.5. *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.

Although 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 [27]. He defines intricate syntax and semantics for the 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 theorems are directly proved by induction on typing derivation.

Theorem 2.6. *Non-linear Substitution. For $\Gamma_1, x :_U A \vdash m : B$ and $\Gamma_2 \vdash n : A$, if $|\Gamma_2|$ and merge $\Gamma_1\ \Gamma_2\ \Gamma$ are valid for some Γ, then $\Gamma \vdash m[n/x] : B[n/x]$.*

Theorem 2.7. *Linear Substitution. For $\Gamma_1, x :_L A \vdash m : B$ and $\Gamma_2 \vdash n : A$, if merge $\Gamma_1\ \Gamma_2\ \Gamma$ is valid for some Γ, then $\Gamma \vdash m[n/x] : B[n/x]$.*

Notice in Theorem 2.6 of non-linear substitutions, there exists an extra condition $|\Gamma_2|$ which Theorem 2.7 of linear substitutions lacks. This condition ensures that linear variables do not occur within the substituted term, thus cannot be duplicated by substitution.

2.8.4 Type Soundness.

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

Theorem 2.8. *Validity. 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, validity and various inversion lemmas proven, subject reduction can now be proved by induction on typing derivation.

Theorem 2.9. *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.

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 from CLC to CCω. We assume familiarity with CCω syntax here and define the erasure function as follows.

Definition 2.13.

$$\begin{aligned}
[\![x]\!] &= x \\
[\![U_i]\!] &= Type_i \\
[\![L_i]\!] &= Type_i \\
[\![(x :_s A) \to B]\!] &= (x : [\![A]\!]) \to [\![B]\!] \\
[\![(x :_s A) \multimap B]\!] &= (x : [\![A]\!]) \to [\![B]\!] \\
[\![\lambda x :_s A.n]\!] &= \lambda x : [\![A]\!].[\![n]\!] \\
[\![m\ n]\!] &= [\![m]\!]\ [\![n]\!]
\end{aligned}$$

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

Definition 2.14.

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

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

Theorem 2.15. *Reduction Erasure. For any CLC terms m and n, if there is $m \rightsquigarrow_{CLC} n$, then there is $[\![m]\!] \rightsquigarrow_{CC\omega} [\![n]\!]$.*

Theorem 2.16. *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 2.15 and 2.16. This is contradictory to the strong normalization property of CCω proven through the Girard-Tait method [13], so this hypothetical term does not exist in CLC.

Theorem 2.17. *Well-typed CLC terms are strongly normalizing.*

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

Definition 2.18.

$$\begin{aligned}
(\!|x|\!) &= x \\
(\!|Type_i|\!) &= U_i \\
(\!|(x : A) \to B|\!) &= (x :_U (\!|A|\!)) \to (\!|B|\!) \\
(\!|\lambda x : A.n|\!) &= \lambda x :_U (\!|A|\!).(\!|n|\!) \\
(\!|m\ n|\!) &= (\!|m|\!)\ (\!|n|\!)
\end{aligned}$$

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

Definition 2.19.

$$\begin{aligned}
(\!|\epsilon|\!) &= \epsilon \\
(\!|\Gamma, x : A|\!) &= (\!|\Gamma|\!), x :_U (\!|A|\!)
\end{aligned}$$

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

Theorem 2.20. *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 2.21. *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 domains of function types are annotated with sort U. In practice, these annotations can often be omitted if algorithmic type checking is able to infer them from context.

3 The Extension to CILC

Although the core CLC language is extremely expressive, formally shown by Theorem 2.21 to be at least as expressive as the predicative CCω, the complexity of Church-style encodings to do so will quickly become unmanageable. To address this, we extend CLC with rules for defining inductive linear types in the style of CIC.

We have formalized CILC in our Coq development and proven its soundness.

3.1 Syntax of CILC

Figure 8. Syntax

$$
\begin{array}{lcll}
C, I, P, Q, f & ::= & \ldots & \text{expressions} \\
& | & \text{Ind}_s(X : A)\{C_1 \mid \ldots \mid C_k\} & \\
& | & \text{Constr}(k, I) & \\
& | & \text{Case}(m, Q)\{f_1 \mid \ldots \mid f_k\} & \\
& | & \text{DCase}(m, Q)\{f_1 \mid \ldots \mid f_k\} & \\
& | & \text{Fix } f : A := m &
\end{array}
$$

As shown in Figure 8, CILC extends CIC with syntax for the introduction and elimination of inductive types. Ind is used during the formation of new inductive types, and Constr is used for introducing their constructors. Case and DCase are non-dependent and dependent case eliminators respectively. CIC-style primitive recursion has been factored out of case eliminators in favor of a Coq-style Fix construct. The flexibility of an independent Fix construct allows for a cleaner formalization of inductive linear type elimination.

3.2 Arity

For type A and sort s, the judgment $arity(A, s)$ is inductively defined over the structure of A as depicted in Figure 9.

Figure 9. Arity

$$\frac{}{arity(s_i, s)}\text{A-Sort} \qquad \frac{arity(A, s)}{arity((x :_U M) \to A, s)}\text{A}\to$$

Definition 3.1. For some arity type A and sort s', the term $A\langle s'\rangle$ is inductively defined over the structure of A as follows.

$$((x :_U M) \to A)\langle s'\rangle = (x :_U M) \to A\langle s'\rangle$$
$$(s)\langle s'\rangle = s'$$

Definition 3.2. For some arity type A, term I and sort s', the term $A\{I, s'\}$ is inductively defined over the structure of A as follows.

$$((x :_U M) \to A)\langle I, s'\rangle = (x :_U M) \to A\langle (I\ x), s'\rangle$$
$$(s)\langle I, s'\rangle = (\_ :_U I) \to s'$$

3.3 Strict Positivity

For type P and variable X, the judgment $postive(P, X)$ is inductively defined over the structure of P as depicted in Figure 10.

We say that X occurs strictly positive in P if the judgment $postive(P, X)$ is valid.

Figure 10. Strict Positivity

$$\frac{(\forall i = 1 \ldots k) \qquad X \notin m_i}{positive((X\ m_1 \ldots m_k), X)}\text{Pos-X}$$

$$\frac{X \notin M \qquad positive(P, X)}{positive((x :_s M) \to P, X)}\text{Pos}\to$$

$$\frac{X \notin M \qquad positive(P, X)}{positive((x :_s M) \multimap P, X)}\text{Pos}\multimap$$

3.4 Non-Linear Constructor

For type C and variable X, the judgment $constructor_U(C, X)$ is inductively defined over the structure of C as shown in Figure 11.

Non-linear constructors are used for constructing non-linear objects that may be freely used. These constructors may not be applied to restricted terms as this may cause duplication.

Figure 11. Non-linear Constructor

$$\frac{(\forall i = 1 \ldots k) \qquad X \notin m_i}{constructor_U((X\ m_i \ldots m_k), X)}\text{U-Constr-X}$$

$$\frac{positive(P, X) \qquad constructor_U(C, X)}{constructor_U((\_ :_U P) \to C, X)}\text{U-Constr-pos}$$

$$\frac{X \notin M \qquad constructor_U(C, X)}{constructor_U((x :_U M) \to C, X)}\text{U-Constr}\to$$

3.5 Linear Constructor

For type C and variable X, the judgment $constructor_L(C, X)$ is inductively defined over the structure of C as shown in Figure 12.

Linear constructors are used for constructing linear objects that must be used once. These constructors may be applied to linear terms as restricted usage prevents duplication and enforces single usage.

An unapplied linear constructor is a non-linear entity as they can be created freely "out of thin air". However, once a linear constructor has been partially applied to a linear term, its linearity is "activated" in rules L-Constr-Pos-2 and L-Constr-2→, essentially forcing the rest of its arguments to be fully applied. This is to prevent partially applied constructors from duplicating its linear arguments.

Figure 12. Linear Constructor

$$\frac{(\forall i = 1...k) \qquad X \notin m_i}{constructor_L((X\ m_i...m_k), X)}\text{L-Constr-X}$$

$$\frac{positive(P, X) \qquad constructor_L(C, X)}{constructor_L((\_ :_U P) \to C, X)}\text{L-Constr-Pos-1}$$

$$\frac{X \notin M \qquad constructor_L(C, X)}{constructor_L((x :_U M) \to C, X)}\text{L-Constr-1}\to$$

$$\frac{positive(P, X) \qquad activation(C, X)}{constructor_L((\_ :_L P) \to C, X)}\text{L-Constr-Pos-2}$$

$$\frac{X \notin M \qquad activation(C, X)}{constructor_L((x :_L M) \to C, X)}\text{L-Constr-2}\to$$

For type C and variable X, the judgment $activation(C, X)$ is inductively defined over the structure of C as shown in Figure 13. Intuitively, $activation(C, X)$ ensures the linearity of C.

Figure 13. Activation

$$\frac{(\forall i = 1...k) \qquad X \notin m_i}{activation((X\ m_i...m_k), X)}\text{Act-X}$$

$$\frac{positive(P, X) \qquad activation(C, X)}{activation((\_ :_P) \multimap C, X)}\text{Act-Pos}$$

$$\frac{X \notin M \qquad activation(C, X)}{activation((x :_M) \multimap C, X)}\text{Act}\multimap$$

3.6 Introduction Rules

The formation of new inductive types and their constructors are presented in Figure 14.

Figure 14. Inductive Type Formation

$$\frac{(\forall j = 1...n) \qquad arity(A, s) \qquad constructor_s(C_j, X) \qquad |\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma, X :_U A \vdash C_j : t_i}{\Gamma \vdash \text{Ind}_s(X : A)\{C_1|...|C_k\} : A}\text{Ind}$$

$$\frac{1 \le i \le k \qquad |\Gamma| \qquad I := \text{Ind}_s(X : A)\{C_1|...|C_k\} \qquad \Gamma \vdash I : A}{\Gamma \vdash \text{Constr}(i, I) : C_i[I/X]}\text{Constr}$$

3.7 Non-Dependent Case

Definition 3.3. For constructor type C and variables X and Q, the term $C\{X, Q\}$ is inductively defined over the structure of C as follows.

$$\begin{aligned}((\_ :_s P) \to C)\{X, Q\} &= (\_ :_s P) \multimap C\{X, Q\}\\ ((\_ :_s P) \multimap C)\{X, Q\} &= (\_ :_s P) \multimap C\{X, Q\}\\ ((x :_s M) \to C)\{X, Q\} &= (x :_s P) \multimap C\{X, Q\}\\ ((x :_s M) \multimap C)\{X, Q\} &= (x :_s P) \multimap C\{X, Q\}\\ (X\ m_1...m_k)\{X, Q\} &= (Q\ m_1...m_k)\end{aligned}$$

We define the notation $C[I, P] := C\{X, Q\}[I/X, P/Q]$.

The rule for non-dependent case elimination is presented in Figure 15. This elimination rule is non-dependent because the motive Q may not depend on the discriminated term m itself.

Figure 15. Non-dependent Case Elimination

$$\frac{(\forall i = 1...k_1) \qquad merge\ \Gamma_1\ \Gamma_2\ \Gamma \qquad arity(A, s) \qquad I := \text{Ind}_s(X : A)\{C_1|...|C_{k_1}\} \qquad \overline{\Gamma_2} \vdash Q : A\langle s'\rangle \qquad \Gamma_1 \vdash m : (I\ a_1...a_{k_2}) \qquad \Gamma_2 \vdash f_i : C_i[I, Q]}{\Gamma \vdash \text{Case}(m, Q)\{f_1|...|f_{k_1}\} : (Q\ a_1...a_{k_2})}\text{Case}$$

3.8 Dependent Case

Definition 3.4. For some non-linear constructor type C and type variables X, Q and c, the term $C\{X, Q, c\}$ is inductively defined over the structure of C as follows.

$$\begin{aligned}((\_ :_U P) \to C)\{X, Q, c\} &= (p :_U P) \multimap C\{X, Q, (c\ p)\}\\ ((x :_U M) \to C)\{X, Q, c\} &= (x :_U M) \multimap C\{X, Q, (c\ x)\}\\ (X\ m_1...m_n)\{X, Q, c\} &= (Q\ m_1...m_n\ c)\end{aligned}$$

We define the notation $C[I, P, t] := C\{X, Q, c\}[I/X, P/Q, t/c]$.

Dependent elimination of non-linear inductive objects is presented in Figure 16. This elimination rule is dependent because the motive Q may depend on the discriminated term m. For this very reason, m is not allowed to have linear type as doing so may transport restricted variable into types.

Figure 16. Dependent Case Elimination

$$\frac{|\Gamma_1| \quad (\forall i = 1...n) \quad merge\ \Gamma_1\ \Gamma_2\ \Gamma \quad arity(A, U_i) \quad I := \mathrm{Ind}_U(X : A)\{C_1|...|C_n\} \quad \overline{\Gamma_2} \vdash Q : A\langle I, s'\rangle \quad \Gamma_1 \vdash m : (I\ a_1...a_n) \quad \Gamma_2 \vdash f_i : C_i[I, Q, Constr(i, I)]}{\Gamma \vdash \mathrm{DCase}(m, Q)\{f_1|...|f_n\} : (Q\ a_1...a_n\ m)}\ \textsc{DCase}$$

3.9 Fixpoint

The typing rule of fix-points presented in Figure 17 is mostly standard. A syntactic guard condition is utilized to conservatively ensure termination. The fix-point term must have non-linear type in a pure context, because recursive calls may cause duplicated usage of restricted variables.

Figure 17. Fixpoint

$$\frac{|\Gamma| \quad \Gamma \vdash A : U_i \quad \Gamma, f :_U A \vdash m : A \quad \text{guard condition}}{\Gamma \vdash \mathrm{Fix}\ f : A := m : A}\ \textsc{fix}$$

3.10 Extended Reduction

As the reduction semantics for the extended language is standard [19], we only present the most interesting ι-reduction cases.

Figure 18. ι-Reductions

$$\frac{}{\mathrm{Case}((\mathrm{Constr}(i, I)\ a_1...a_n), Q)\{f_1|...f_k\} \rightsquigarrow (f_i\ a_1...a_k)}\ \textsc{Step-Case-}\iota$$

$$\frac{}{\mathrm{DCase}((\mathrm{Constr}(i, I)\ a_1...a_k), Q)\{f_1|...f_k\} \rightsquigarrow (f_i\ a_1...a_k)}\ \textsc{Step-DCase-}\iota$$

$$\frac{}{\mathrm{Fix}\ f : A := m \rightsquigarrow m[(\mathrm{Fix}\ f : A := m)/f]}\ \textsc{Step-Fix-}\iota$$

3.11 Non-Linear Connectives

Non-linear inductive types have been extensively studied since their inception. We will omit discussion of non-linear inductive types here as all inductive definitions in the predicative fragment of CIC are available in CILC.

3.12 Linear Connectives

In Figure 19, we demonstrate how the $\otimes$ and $\oplus$ connectives of linear logic could be defined in CILC as inductive linear types. For any term with inductive linear type, linearity will ultimately force the term to be eliminated by a Case expression.

Figure 19. Inductively Defined Linear Connectives

$$\begin{aligned}
&_ \otimes _ := \mathrm{Ind}_L(X : (A :_U L_i) \to (B :_U L_i) \to L_i)\{\\
&\quad | \ (A :_U L_i) \to (B :_U L_i)\\
&\qquad \to (_ :_L A) \multimap (_ :_L B) \multimap (X\ A\ B)\\
&\quad \}\\
&\langle _, _ \rangle := \mathrm{Constr}(1, \otimes)\\
&\\
&_ \oplus _ := \mathrm{Ind}_L(X : (A :_U L_i) \to (B :_U L_i) \to L_i)\{\\
&\quad | \ (A :_U L_i) \to (B :_U L_i) \to (_ :_L A) \multimap (X\ A\ B)\\
&\quad | \ (A :_U L_i) \to (B :_U L_i) \to (_ :_L B) \multimap (X\ A\ B)\\
&\quad \}\\
&\mathrm{inl}(_) := \mathrm{Constr}(1, \oplus)\\
&\mathrm{inr}(_) := \mathrm{Constr}(2, \oplus)
\end{aligned}$$

3.13 Mixed Connectives

In Figure 20, we introduce a Σ_L connective. Σ_L is similar to the standard Σ type where the type of the second component is dependent on the value of the first component. The key difference between these two type constructors is that Σ's second component is non-linear, whereas Σ_L's second component is linear.

Figure 20. Mixed Connective

$$\begin{aligned}
&\Sigma_L := \mathrm{Ind}_L(X : (A :_U U_i) \to (B :_U (_ :_U A) \to L_i) \to L_i)\{\\
&\quad | \ (A :_U U_i) \to (B :_U (_ :_U A) \to L_i)\\
&\qquad \to (m :_U A) \to (_ :_L B\ m) \multimap (X\ A\ B)\\
&\quad \}\\
&[_, _] := \mathrm{Constr}(1, \Sigma_L)
\end{aligned}$$

4 Examples

In this section we will demonstrate some applications of CILC in practice. When it is clear from context whether a type is linear or non-linear, we will omit writing sort annotations on function types. We also use Coq style concrete syntax from our implementation for the sake of readability. The full code for each example is available with our implementation.

4.1 Stateful Reasoning

We first postulate five state axioms. These axioms could be viewed as an interface to a trusted memory allocator. Using these axioms, we will build safe random access arrays from the ground up.

Definition 4.1. State axioms.

- $\mapsto : \mathbb{N} \to (A : U) \to L$
  The $\mapsto$ is an infix linear type constructor. It fulfills a role similar to L3's [16] memory access capability, or Separation Logic's [20] points-to assertion. Intuitively, a capability $(l \mapsto A)$ is a proof that a term of non-linear type A is currently stored at address l.
- **new** : $(A : U) \to A \to \Sigma_L l : \mathbb{N}.(l \mapsto A)$
  `new` is a function that allocates a new memory cell for a term non-linear type A and initializes the cell with its argument. This returns an address l paired with a capability $(l \mapsto A)$.
- **free** : $(A : U) \to (l : \mathbb{N}) \to (l \mapsto A) \to$ `unit`
  `free` is a function that de-allocates the memory cell at location l which is currently in use. The capability $(l \mapsto A)$ is consumed so the memory cell can no longer be accessed.
- **get** : $(A : U) \to (l : \mathbb{N}) \to (l \mapsto A) \to \Sigma_{L\_} : A.(l \mapsto A)$
  `get` is a function that retrieves the term stored at address l when given a proof $(l \mapsto A)$ there is indeed a term of type A currently stored there. The type of `get`'s output is the rather complicated $\Sigma_{L\_} : A.(l \mapsto A)$. This not only returns the term stored at l, but also a proof $(l \mapsto A)$ that the memory state has not changed and may be accessed again.
- **set** : $(A\ B : U) \to (l : \mathbb{N}) \to B \to (l \mapsto A) \to (l \mapsto B)$
  `set` has the most complicated type of all the axioms presented here. Intuitively, if address l is allocated and storing some term of type A, `set` may put a term type B into the the store at address l, overwriting the original term. This is the so called "strong update" operation. The consumption of $(l \mapsto A)$ and return of $(l \mapsto B)$ accurately characterize this update process.

Due to the fact that $(l \mapsto A)$ is an abstract 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, $(l \mapsto A)$ itself is a first-class object in CILC, meaning that it can be the result of computations (large eliminations), input to computations and storage in data structures, etc.

A random access array is a contiguous region of memory that can be accessed by an address pointing to the start of this region and an offset. In Figure 21, we declare a linear inductive `ArrVec` to track the memory used by the array. `ArrVec` is parameterized by the type A of elements stored in the array and the address l of the initial element. `ArrVec` also takes an additional argument of type $\mathbb{N}$ indicating the length of the array.

Figure 21. Proof of Arrays

```
Inductive ArrVec (A : U) (l : ℕ) : ℕ → L :=
| Nil : ArrVec A l 0
| Cons : (n : ℕ) → ((l + n) ↦ A) →
    ArrVec A l n → ArrVec A l (S n).
```

Notice that for the `Cons` constructor, each time a proof of $(l + n \mapsto A)$ is used to extend the current `ArrVec`, the next application of `Cons` expects a proof of $(l + S\ n \mapsto A)$. So for some type A and natural numbers l, n, a term of type `ArrVec` $A\ l\ n$ is a proof that a contiguous region of memory starting from address l can be accessed by any offset less than n. Arrays can now be characterized as an address l paired with an array proof `ArrVec` $A\ l\ n$ as shown in Figure 22.

Figure 22. Arrays

```
Definition Array (A : U) (n : ℕ) : L :=
  Σ_L l : ℕ.ArrVec A l n.
```

To manipulate arrays, we construct a recursive helper function whose signature is shown in Figure 23. When given an array proof `ArrVec` $A\ l\ n$, an offset m and proof pf such that $m < n$, `nth` will return a capability $(l + m \mapsto A)$ along with a function $(l + m \mapsto A) \multimap$ `ArrVec` $A\ l\ n$. The capability $(l + m \mapsto A)$ grants the ability to manipulate the store at offset m using the state axioms. The function $(l + m \mapsto A) \multimap$ `ArrVec` $A\ l\ n$ is constructed using the rest of the capabilities in `ArrVec` $A\ l\ n$ that are not at offset m. Intuitively, $(l + m \mapsto A) \multimap$ `ArrVec` $A\ l\ n$ is the original array proof with a missing capability at offset m which can be restored by applying capability $(l + m \mapsto A)$.

Figure 23. ArrVec Traversal

```
Fixpoint nth
  (A : U)
  (l m n : ℕ)
  (pf : m < n)
  (v : ArrVec A l n)
: (l + m ↦ A) ⊗ ((l + m ↦ A) ⊸ ArrVec A l n).
```

An indexing function can now be constructed as a wrapper for `nth` as shown in Figure 24. Given an array `Array` $A\ n$, an offset m and proof that $m < n$, `nth` can be used to extract the capability at offset m, allowing `get` to retrieve the stored value. Finally, the array proof is restored and returned alongside the indexing result. This definition of `index` is

Figure 24. Array Indexing

```
Definition index
  (A : U)
  (m n : ℕ)
  (pf : m < n)
  (a : Array A n)
: Σ_L _ : A. Array A n
```

safe because the proof $m < n$ ensures that offsets used for indexing are always within bounds.

From our experiments with proof erasure, we observe that erasing the proof arguments of `index` in a reasonable way can result in a single call to `get`. This indicates that the arrays described here can be compiled for constant time access.

4.2 Protocol Enforcement

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

In Figure 25, 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 25. 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 : `session`) $\to$ `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 have finished, `close` will close and reclaim the channel.
- **send** : $(A : U) \to A \to$ (ss : `session`) $\to$ `channel` (`SEND` A ss) $\to$ `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 : `session`) $\to$ `channel` (`RECV` A ss) $\to$ $F\_$: A.`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 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.

4.3 Mutable Data

Figure 26 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 $A \multimap \mathit{list}\ A \multimap \mathit{list}\ A$. Each `_cons_` is 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 ($\mathit{list}\ A \multimap \mathit{list}\ A$), indicating that k must be applied exactly once. Using linear types for continuations is surprisingly accurate as control-flow itself cannot be freely duplicated nor discarded.

5 Related Work

Linear types are a class of type systems inspired by Girard's substructural Linear Logic [10]. 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 [25, 26] then applies an analogous restriction to variable usage in simple type theory, leading to the development of linear type theory where terms respect resource. A term calculus for linear type theory is 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, 16, 28] featuring linear types have also been

Figure 26. 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.
```

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, 11], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár [24] presents a linear dependent type theory, with syntax and semantics drawing inspiration from DILL [3]. Krishnaswami et al. present a dependent linear type theory [12] 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 [17]. Luo et al. [14] 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 [15], Atkey's Quantitative Type Theory (QTT) [2] uses semi-ring annotations to track variable occurrence, simulating irrelevance and linear and affine types within a unified framework. The Idris 2 programming language [6] implements QTT as its core type system.

6 Future Work

In early versions of CLC, an impredicative *Prop* universe sat at the bottom of the universe hierarchy. This has since been removed due to the additional difficulty of working with *Prop* in the soundness proof of CILC. We believe that these difficulties are not intrinsic and *Prop* could be added back with careful proof work.

We aim to fully verify the typechecking algorithm employed by the current implementation. The unification algorithm used for implicit argument resolution is extremely ad-hoc. As the problem of unification for linear types has not been thoroughly studied, we hope to improve this situation as well.

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., and Plotkin, G. Dual intuitionistic linear logic. Tech. rep., Labrartory of Compute Science, Univerity of Edinburgh, 1996.

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

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

[13] Luo, Z. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., USA, 1994.

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

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

[16] 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.

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

[18] Norell, U. Towards a practical programming language based on dependent type theory, 2007.

[19] Paulin-Mohring, C. Inductive definitions in the system coq rules and properties. In *Typed Lambda Calculi and Applications* (Berlin, Heidelberg, 1993), M. Bezem and J. F. Groote, Eds., Springer Berlin Heidelberg, pp. 328–345.

[20] 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.

[21] 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.

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

[23] The Coq Development Team. *The Coq Proof Assistant, version 8.11.0*, jan 2020.

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

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

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

[27] Wadler, P. There's no substitute for linear logic.
[28] Xi, H. Applied type system: An approach to practical programming with theorem-proving. *CoRR abs/1703.08683* (2017).

Appendix

A Syntax of CLC

$$
\begin{array}{llll}
i & ::= & 0 \mid 1 \mid 2 \ldots & \text{universe levels} \\
s, t & ::= & U \mid L & \text{sorts} \\
m, n, A, B, M & ::= & U_i \mid L_i \mid x & \text{expressions} \\
& \mid & (x :_s A) \to B & \\
& \mid & (x :_s A) \multimap B & \\
& \mid & \lambda x :_s A.n & \\
& \mid & m\ n &
\end{array}
$$

B Reduction and Equality of CLC

$$\frac{m_1 \rightsquigarrow^* n \qquad m_2 \rightsquigarrow^* n}{m_1 \equiv m_2 : A}\textsc{Join}$$

$$\frac{}{(\lambda x :_s A.m)\ n \rightsquigarrow m[n/x]}\textsc{Step-}\beta$$

$$\frac{A \rightsquigarrow A'}{\lambda x :_s A.m \rightsquigarrow \lambda x :_s A'.m}\textsc{Step-}\lambda\textsc{L}$$

$$\frac{m \rightsquigarrow m'}{\lambda x :_s A.m \rightsquigarrow \lambda x :_s A.m'}\textsc{Step-}\lambda\textsc{R}$$

$$\frac{A \rightsquigarrow A'}{(x :_s A) \to B \rightsquigarrow (x :_s A') \to B}\textsc{Step-L}\to$$

$$\frac{B \rightsquigarrow B'}{(x :_s A) \to B \rightsquigarrow (x :_s A) \to B'}\textsc{Step-R}\to$$

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

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

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

C Confluence of CLC

C.1 Parallel Reduction

To prove the confluence property of CLC, we employ the standard technique utilizing parallel reductions.

$$\frac{}{x \rightsquigarrow_p x}\textsc{PStep-Var} \qquad \frac{}{s_i \rightsquigarrow_p s_i}\textsc{PStep-Sort}$$

$$\frac{A \rightsquigarrow_p A' \qquad m \rightsquigarrow_p m'}{\lambda x :_s A.m \rightsquigarrow_p \lambda x :_s A'.m'}\textsc{PStep-}\lambda$$

$$\frac{m \rightsquigarrow_p m' \qquad n \rightsquigarrow_p n'}{m\ n \rightsquigarrow_p m'\ n'}\textsc{PStep-App}$$

$$\frac{m \rightsquigarrow_p m' \qquad n \rightsquigarrow_p n'}{(\lambda x :_s A.m)\ n \rightsquigarrow_p m'[n'/x]}\textsc{PStep-}\beta$$

$$\frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{(x :_s A) \to B \rightsquigarrow_p (x :_s A') \to B'}\textsc{PStep}\to$$

$$\frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{(x :_s A) \multimap B \rightsquigarrow_p (x :_s A') \multimap B'}\textsc{PStep}\multimap$$

C.2 Reduction Lemmas

Here, we prove some simple lemmas concerning $\rightsquigarrow$, $\rightsquigarrow^*$ and substitution.

Definition C.1. For a term m and a map σ from variables to terms, let $m[\sigma]$ be the term obtained by applying σ uniformly to all free variables in m.

Definition C.2. For maps σ, τ from variables to terms, we say that σ reduces to τ if for any variable x there exists a reduction $(\sigma\ x) \rightsquigarrow^* (\tau\ x)$. We write $\sigma \rightsquigarrow^* \tau$ when it is clear from context that σ, τ are maps and not terms.

Lemma C.3. *For terms m, n and a map σ from variables to terms, if there exist a step $m \rightsquigarrow n$, then there exists a step $m[\sigma] \rightsquigarrow n[\sigma]$.*

Proof. By induction on the derivation of $m \rightsquigarrow n$. □

Lemma C.4. *For terms m_1, m_2, n_1, n_2, if these exists reductions $m_1 \rightsquigarrow^* m_2$ and $n_1 \rightsquigarrow^* n_2$, then there exists reduction $(m_1\ n_1) \rightsquigarrow^* (m_2\ n_2)$.*

Proof. By transitivity of $\rightsquigarrow^*$ and applying rules Step-AppL, Step-AppR. □

Lemma C.5. *For terms A_1, A_2, m_1, m_2 and sort s, if there exists reductions $A_1 \rightsquigarrow^* A_2$ and $m_1 \rightsquigarrow^* m_2$, then there exists reduction $\lambda x :_s A_1.m_1 \rightsquigarrow^* \lambda x :_s A_2.m_2$.*

Proof. By transitivity of $\rightsquigarrow^*$ and applying rules Step-λL, Step-λR. □

Lemma C.6. *For terms A_1, A_2, B_1, B_2 and sort s, if there exists reductions $A_1 \rightsquigarrow^* A_2$ and $B_1 \rightsquigarrow^* B_2$, then there exists reduction $(x :_s A_1) \to B_1 \rightsquigarrow^* (x :_s A_2) \to B_2$.*

Proof. By transitivity of $\rightsquigarrow^*$ and applying rules Step-L$\to$, Step-R$\to$. □

Lemma C.7. *For terms A_1, A_2, B_1, B_2 and sort s, if there exists reductions $A_1 \rightsquigarrow^* A_2$ and $B_1 \rightsquigarrow^* B_2$, then there exists reduction $(x :_s A_1) \multimap B_1 \rightsquigarrow^* (x :_s A_2) \multimap B_2$.*

Proof. By transitivity of $\rightsquigarrow^*$ and applying rules Step-L$\multimap$, Step-R$\multimap$. □

Lemma C.8. *For terms m, n and a map σ from variables to terms, if there exist a reduction $m \rightsquigarrow^* n$, then there exists a reduction $m[\sigma] \rightsquigarrow^* n[\sigma]$.*

Proof. By induction on the derivation of $\rightsquigarrow^*$, the transitivity of $\rightsquigarrow^*$ and Lemma C.3. □

Lemma C.9. *For maps σ, τ from variables to terms, if there is a map reduction $\sigma \rightsquigarrow^* \tau$, then for any term m these is a reduction $m[\sigma] \rightsquigarrow^* m[\tau]$.*

Proof. By induction on the structure of m, applying Lemmas C.4, C.5, C.6, C.7. □

C.3 Equality Lemmas

Here, we prove some simple lemmas concerning $\rightsquigarrow^*$, $\equiv$ and substitution.

Definition C.10. For maps σ, τ from variables to terms, we say that σ is equal to τ if for any variable x there exists an equality $(\sigma\ x) \equiv (\tau\ x)$. We write $\sigma \equiv \tau$ when it is clear from context that σ, τ are maps and not terms.

Lemma C.11. *For any map f from terms to terms, if for any terms m, n such that $m \rightsquigarrow n$ implies $f\ m \equiv f\ n$, then for any terms m, n equality $m \equiv n$ implies $f\ m \equiv f\ n$.*

Proof. By the properties of the transitive reflexive closure $\rightsquigarrow^*$ and that $\equiv$ is an equivalence relation. □

Lemma C.12. *For terms m_1, m_2, n_1, n_2, if there exists equalities $m_1 \equiv m_2$ and $n_1 \equiv n_2$, then there exists equality $(m_1\ n_1) \equiv (m_2\ n_2)$.*

Proof. By transitivity of $\equiv$ and applying rules Join, Step-AppL, Step-AppR. □

Lemma C.13. *For terms A_1, A_2, m_1, m_2 and sort s, if there exists equalities $A_1 \equiv A_2$ and $m_1 \equiv m_2$, then there exists equality $\lambda x :_s A_1.m_1 \equiv \lambda x :_s A_2.m_2$.*

Proof. By transitivity of $\equiv$ and applying rules Join, Step-λL, Step-λR. □

Lemma C.14. *For terms A_1, A_2, B_1, B_2 and sort s, if there exists equalities $A_1 \equiv A_2$ and $B_1 \equiv B_2$, then there exists equality $(x :_s A_1) \to B_1 \equiv (x :_s A_2) \to B_2$.*

Proof. By transitivity of $\equiv$ and applying rules Join, Step-L$\to$, Step-R$\to$. □

Lemma C.15. *For terms A_1, A_2, B_1, B_2 and sort s, if there exists equalities $A_1 \equiv A_2$ and $B_1 \equiv B_2$, then there exists equality $(x :_s A_1) \multimap B_1 \equiv (x :_s A_2) \multimap B_2$.*

Proof. By transitivity of $\equiv$ and applying rules Join, Step-L$\multimap$, Step-R$\multimap$. □

Lemma C.16. *For terms m, n and map σ from variables to terms, if there is equality $m \equiv n$, then there is equality $m[\sigma] \equiv n[\sigma]$.*

Proof. By Lemmas C.11 and C.3. □

Lemma C.17. *For maps σ, τ from variables to terms and term m, if these is map equality $\sigma \equiv \tau$, then there is equality $m[\sigma] \equiv m[\tau]$.*

Proof. By induction on the structure of m, applying Lemmas C.12, C.13, C.14, C.15. □

Lemma C.18. *For terms m_1, m_2, n, if there is equality $m_1 \equiv m_2$, then there is equality $n[m_1/x] \equiv n[m_2/x]$ for any variable $x \in FV(n)$.*

Proof. This is a special case of Lemma C.17 where σ maps x to m_1 and τ maps x to m_2. □

C.4 Parallel Reduction Lemmas

Definition C.19. For maps σ, τ from variables to terms, we say σ parallel reduces to τ if for any variable x there exists a parallel reduction $(\sigma\ x) \rightsquigarrow_p (\tau\ x)$. We write $\sigma \rightsquigarrow_p \tau$ when it is clear from context that σ, τ are maps and not terms.

Lemma C.20. *For any term m, there exists a reflexive parallel reduction $m \rightsquigarrow_p m$.*

Proof. By induction on the structure of m. □

Lemma C.21. *For any map σ from variables to terms, there exists a reflexive parallel map reduction $\sigma \rightsquigarrow_p \sigma$.*

Proof. By Definition C.19 and Lemma C.20. □

Lemma C.22. *For any terms m, n, if there exists step $m \rightsquigarrow n$, then there exists a parallel reduction $m \rightsquigarrow_p n$.*

Proof. By induction on the derivation of $m \rightsquigarrow n$ and Lemma C.20. □

Lemma C.23. *For terms m, n, if there exists parallel reduction $m \rightsquigarrow_p n$, then there exists a reduction $m \rightsquigarrow^* n$.*

Proof. By induction on the derivation of $m \rightsquigarrow_p n$, utilizing the transitive property of $\rightsquigarrow^*$ and Lemmas C.4, C.5, C.6, C.7, C.8, C.9. □

Lemma C.24. *For terms m, n and map σ from variables to terms, if there exists parallel reduction $m \rightsquigarrow_p n$, there exists parallel reduction $m[\sigma] \rightsquigarrow_p n[\sigma]$.*

Proof. By induction on the derivation of $m \rightsquigarrow_p n$ and Lemma C.20. □

Lemma C.25. *For terms m, n and maps σ, τ from variables to terms, if there exists parallel reduction $m \rightsquigarrow_p n$ and parallel map reduction $\sigma \rightsquigarrow_p \tau$, there exists parallel reduction $m[\sigma] \rightsquigarrow_p n[\tau]$.*

Proof. By induction on the derivation of $m \rightsquigarrow_p n$. □

Lemma C.26. *For terms m_1, m_2, n, if there is parallel reduction $m_1 \rightsquigarrow_p m_2$, then there is parallel reduction $n[m_1/x] \rightsquigarrow_p n[m_2/x]$ for any variable $x \in FV(n)$.*

Proof. By Lemma C.20, this is a special case of Lemma C.25 where σ maps x to m_1 and τ maps x to m_2. □

C.5 Confluence Theorem

We first show that $\rightsquigarrow_p$ satisfies the diamond property. Using the diamond property, we ultimately prove the confluence theorem.

Lemma C.27. *CLC term reduction has the diamond property. For terms m, m_1, m_2, if there are parallel reductions $m \rightsquigarrow_p m_1$ and $m \rightsquigarrow_p m_2$, then there exists term m' such that $m_1 \rightsquigarrow_p m'$ and $m_2 \rightsquigarrow_p m'$.*

Proof. By induction on the derivation of $m \rightsquigarrow_p m_1$. Each case in the induction specializes m appearing in $m \rightsquigarrow_p m_2$, allowing one to invert its derivation in a syntax directed way and then to apply the induction hypothesis. The difficult cases are due to PSTEP-β as it concerns substitution, so Lemma C.25 is used to push these cases through. □

Lemma C.28. *Strip lemma. For terms m, m_1, m_2, if there is parallel reduction $m \rightsquigarrow_p m_1$ and reduction $m \rightsquigarrow^* m_2$, then there exists term m' such that $m_1 \rightsquigarrow^* m'$ and $m_2 \rightsquigarrow_p m'$.*

Proof. By induction on the derivation of $m \rightsquigarrow_p m_1$, utilizing transitivity of $\rightsquigarrow^*$ and Lemmas C.22, C.23, C.27. □

Theorem C.29. *CLC term reduction is confluent. For terms m, m_1, m_2, if there are reductions $m \rightsquigarrow^* m_1$ and $m \rightsquigarrow^* m_2$, then there exists term m' such that $m_1 \rightsquigarrow^* m'$ and $m_2 \rightsquigarrow^* m'$.*

Proof. By induction on the derivation of $m \rightsquigarrow^* m_1$, utilizing transitivity of $\rightsquigarrow^*$ and Lemmas C.22, C.23, C.28. □

C.6 Corollaries of Confluence

The following results are all corollaries of confluence, proven using a combination of induction, transitivity and confluence. These corollaries allow us to refute false reductions and equalities in future proofs.

Corollary C.30. *For a universe s_i and term m, if there is reduction $s_i \rightsquigarrow^* m$, then $m = s_i$.*

Corollary C.31. *For variable x and term m, if there is reduction $x \rightsquigarrow^* m$, then $m = x$.*

Corollary C.32. *For terms A, B, m and sort s, if there is reduction $(x :_s A) \to B \rightsquigarrow^* m$, then there exists A', B' such that there are reductions $A \rightsquigarrow^* A'$, $B \rightsquigarrow^* B'$ and $m = (x :_s A') \to B'$.*

Corollary C.33. *For terms A, B, m and sort s, if there is reduction $(x :_s A) \multimap B \rightsquigarrow^* m$, then there exists A', B' such that there are reductions $A \rightsquigarrow^* A'$, $B \rightsquigarrow^* B'$ and $m = (x :_s A') \multimap B'$.*

Corollary C.34. *For terms A, m, n and sort s, if there is reduction $\lambda x :_s A.m \rightsquigarrow^* n$, then there exists A', m' such that there are reductions $A \rightsquigarrow^* A'$, $m \rightsquigarrow^* m'$ and $n = \lambda x :_s A'.m'$.*

Corollary C.35. *For sorts s, t and levels i, j, if there is equality $s_i \equiv t_j$, then there is $s = t$ and $i = j$.*

Corollary C.36. *For terms A_1, A_2, B_1, B_2 and sorts s, t, if there is equality $(x :_s A_1) \to B_1 \equiv (x :_t A_2) \to B_2$, then there are equalities $A_1 \equiv A_2$, $B_1 \equiv B_2$ and $s = t$.*

Corollary C.37. *For terms A_1, A_2, B_1, B_2 and sorts s, t, if there is equality $(x :_s A_1) \multimap B_1 \equiv (x :_t A_2) \multimap B_2$, then there are equalities $A_1 \equiv A_2$, $B_1 \equiv B_2$ and $s = t$.*

D Context of CLC

Contexts of CLC are of the form $x_1 :_s 1A_1, x_2 :_s 2A_2, ...x_k :_s kA_k$ where each free variable x_i is assigned a type A_i and sort s_i. Contexts will be referred to by meta variables Γ and Δ.

$$\frac{}{\epsilon \vdash}\text{WF-}\epsilon \qquad \frac{\Gamma \vdash \quad \overline{\Gamma} \vdash A : U_i}{\Gamma, x :_U A \vdash}\text{WF-U}$$

$$\frac{\Gamma \vdash \quad \overline{\Gamma} \vdash A : L_i}{\Gamma, x :_L A \vdash}\text{WF-L}$$

$$\frac{}{|\epsilon|}\text{PURE-}\epsilon \qquad \frac{|\Gamma| \quad \Gamma \vdash A : U_i}{|\Gamma, x :_U A|}\text{PURE-U}$$

$$\frac{}{merge\ \epsilon\ \epsilon\ \epsilon}\text{MERGE-}\epsilon$$

$$\frac{merge\ \Gamma_1\ \Gamma_2\ \Gamma}{merge\ (\Gamma_1, x :_U A)\ (\Gamma_2, x :_U A)\ (\Gamma, x :_U A)}\text{MERGE-U}$$

$$\frac{merge\ \Gamma_1\ \Gamma_2\ \Gamma \qquad x \notin \Gamma_2}{merge\ (\Gamma_1, x :_L A)\ \Gamma_2\ (\Gamma, x :_L A)}\text{MERGE-L1}$$

$$\frac{merge\ \Gamma_1\ \Gamma_2\ \Gamma \qquad x \notin \Gamma_1}{merge\ \Gamma_1\ (\Gamma_2, x :_L A)\ (\Gamma, x :_L A)}\text{MERGE-L2}$$

D.1 Merge Lemmas

Since weakening and contraction rules will not be allowed on restricted variables, it is necessary to have lemmas that enable the manipulation of contexts.

Lemma D.1. *For contexts $\Gamma_1, \Gamma_2, \Gamma$, if there is merge $\Gamma_1\ \Gamma_2\ \Gamma$, then there is merge $\Gamma_2\ \Gamma_1\ \Gamma$.*

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.2. *For any context Γ, if there is $|\Gamma|$, then there is merge $\Gamma\ \Gamma\ \Gamma$.*

Proof. By induction on the derivation of $|\Gamma|$. □

Lemma D.3. *For any context* Γ*, there is merge* $\overline{\Gamma}\ \Gamma\ \Gamma$.

Proof. By induction on the structure of Γ. □

Lemma D.4. *For any context* Γ*, there is merge* $\Gamma\ \overline{\Gamma}\ \Gamma$.

Proof. By induction on the structure of Γ. □

Lemma D.5. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *and* $|\Gamma|$*, then there is* $|\Gamma_1|$ *and* $|\Gamma_2|$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.6. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$ *, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *and* $|\Gamma_1|$*, then there is* $\Gamma = \Gamma_2$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.7. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$ *, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *and* $|\Gamma_2|$*, then there is* $\Gamma = \Gamma_1$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.8. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$*, and also* $|\Gamma_1|$, $|\Gamma_2|$*, then there is* $|\Gamma|$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.9. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$*, and also* $|\Gamma_1|$, $|\Gamma_2|$*, then there is* $\Gamma_1 = \Gamma_2$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.10. *For contexts* $\Gamma_1, \Gamma_2, \Gamma$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$*, then there is* $\overline{\Gamma_1} = \overline{\Gamma}$ *and* $\overline{\Gamma_2} = \overline{\Gamma}$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.11. *For any context* Γ*, there is merge* $\overline{\Gamma}\ \overline{\Gamma}\ \overline{\Gamma}$.

Proof. By induction on the structure of Γ. □

D.2 Restriction and Purity Lemmas

Lemma D.12. *For any context* Γ*, there is* $\overline{\Gamma} = \overline{\overline{\Gamma}}$.

Proof. By induction on the structure of Γ. □

Lemma D.13. *For any context* Γ*, if there is* $|\Gamma|$*, then there is* $\Gamma = \overline{\Gamma}$.

Proof. By induction on the structure of Γ. □

Lemma D.14. *For any context* Γ *, there is* $|\overline{\Gamma}|$.

Proof. By induction on the structure of Γ. □

Lemma D.15. *For any context* Γ*, variable* x *and type* A*, if there is* $x :_U A \in \Gamma$*, then there is* $x :_U A \in \overline{\Gamma}$.

Proof. By induction on the derivation of $x :_U A \in \Gamma$. □

Lemma D.16. *For any context* Γ*, variable* x *and type* A*, there is* $x :_L A \notin \overline{\Gamma}$.

Proof. By induction on the structure of Γ. □

Lemma D.17. *For contexts* $\Gamma_1, \Gamma_2, \Gamma, \Delta_1, \Delta_2$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *and merge* $\Delta_1\ \Delta_2\ \Gamma_1$*, then there exists* Δ *such that merge* $\Delta_1\ \Gamma_2\ \Delta$ *and merge* $\Delta\ \Delta_2\ \Gamma$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma D.18. *For contexts* $\Gamma_1, \Gamma_2, \Gamma, \Delta_1, \Delta_2$*, if there is merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *and merge* $\Delta_1\ \Delta_2\ \Gamma_1$*, then there exists* Δ *such that merge* $\Delta_2\ \Gamma_2\ \Delta$ *and merge* $\Delta_1\ \Delta\ \Gamma$.

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

E Subtyping of CLC

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. $U_0 \preceq U_1 \preceq U_2 \preceq \cdots$
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 $(x :_s A_1) \multimap B_1 \preceq (x :_s A_2) \multimap B_2$

Here, we give an inductive definition of the cumulativity relation ($\preceq$) that is suitable for writing proofs.

$$\frac{}{A \prec A}\ \prec\text{-Refl} \qquad \frac{i_1 \le i_2}{s_{i_1} \prec s_{i_2}}\ \prec\text{-Sort}$$

$$\frac{B_1 \prec B_2}{(x :_s A) \to B_1 \prec (x :_s A) \to B_2}\ \prec\to$$

$$\frac{B_1 \prec B_2}{(x :_s A) \multimap B_1 \prec (x :_s A) \multimap B_2}\ \prec\multimap$$

$$\frac{A' \prec B' \qquad A \equiv A' \qquad B \equiv B'}{A \preceq B}\ \prec\text{-}\preceq$$

E.1 Subtyping Lemmas

Lemma E.1. *For terms* A, B*, if there is* $A \prec B$*, then there is* $A \preceq B$.

Proof. By $\prec$-$\preceq$ and the reflexivity of equality $\equiv$. □

Lemma E.2. *For terms* A, B, C*, if there is* $A \prec B$ *and* $B \equiv C$*, then there is* $A \preceq C$.

Proof. By $\prec$-$\preceq$ and the transitivity of equality $\equiv$. □

Lemma E.3. *For terms* A, B, C*, if there is* $A \equiv B$ *and* $B \prec C$*, then there is* $A \preceq C$.

Proof. By $\prec$-$\preceq$ and the transitivity of equality $\equiv$. □

Lemma E.4. *For terms A, B, if there is $A \equiv B$, then there is $A \preceq B$.*

Proof. By Lemma E.3 and $\prec$-Refl. □

Lemma E.5. *For term A, there is $A \preceq A$.*

Proof. By Lemma E.1 and $\prec$-Refl. □

Lemma E.6. *For natural numbers i, j and sort s such that $i \leq j$, there is $s_i \preceq s_j$.*

Proof. By Lemma E.1 and $\prec$-Sort. □

Lemma E.7. *For terms A, B, C, D, if there is $A \prec B$, $B \equiv C$ and $C \prec D$, then there is $A \preceq D$.*

Proof. By induction on the derivation of $A \prec B$, definition of $\prec$ and Lemmas E.1, E.2, E.3. □

Lemma E.8. *For terms A, B, C, if there is $A \preceq B$ and $B \preceq C$, then there is $A \preceq C$.*

Proof. By transitivity of $\equiv$, rule $\prec$-$\preceq$ and Lemma E.7. □

Lemma E.9. *For sorts s, t and natural numbers i, j, if there is $s_i \preceq t_j$, then there is $s = t$ and $i \leq j$.*

Proof. By transitivity of $\equiv$ and Corollary C.35. □

Lemma E.10. *For terms A_1, A_2, B_1, B_2 and sorts s, t, if there is $(x :_s A_1) \to B_1 \preceq (x :_t A_2) \to B_2$, then there is $A_1 \equiv A_2$ and $B_1 \preceq B_2$ and $s = t$.*

Proof. By transitivity of $\equiv$ and Corollary C.36. □

Lemma E.11. *For terms A_1, A_2, B_1, B_2 and sorts s, t, if there is $(x :_s A_1) \multimap B_1 \preceq (x :_t A_2) \multimap B_2$, then there is $A_1 \equiv A_2$ and $B_1 \preceq B_2$ and $s = t$.*

Proof. By transitivity of $\equiv$ and Corollary C.37. □

Lemma E.12. *For terms A, B and map σ from variables to terms, if there is $A \prec B$, then there is $A[\sigma] \prec B[\sigma]$.*

Proof. By induction on the derivation of $A \prec B$ and the definition of $\prec$. □

Lemma E.13. *For terms A, B and map σ from variables to terms, if there is $A \preceq B$, then there is $A[\sigma] \preceq B[\sigma]$.*

Proof. By rule $\prec$-$\preceq$ and Lemmas C.16, E.12. □

F Typing of CLC

The following rules define well-formed contexts.

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

$$\frac{\Gamma \vdash \qquad \overline{\Gamma} \vdash A : L_i}{\Gamma, x :_L A \vdash}\,\text{L-Ok}$$

The typing rules of CLC are presented below.

$$\frac{|\Gamma|}{\Gamma \vdash s_i : U_{i+1}}\,\text{Sort-Axiom}$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma, x :_U A \vdash B : s_i}{\Gamma \vdash (x :_U A) \to B : U_i}\,\text{U}\to$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_i \qquad \Gamma \vdash B : s_i \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \to B : U_i}\,\text{L}\to$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : U_i \qquad \Gamma, x :_U A \vdash B : s_i}{\Gamma \vdash (x :_U A) \multimap B : L_i}\,\text{U}\multimap$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A : L_i \qquad \Gamma \vdash B : s_i \qquad x \notin \Gamma}{\Gamma \vdash (x :_L A) \multimap B : L_i}\,\text{L}\multimap$$

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash m : A \qquad \overline{\Gamma} \vdash B : s_i \qquad A \preceq B}{\Gamma \vdash m : B}\,\text{Conversion}$$

G Inversion Lemmas of CLC

Lemma G.1. *For any context Γ and terms A, B, s, if there is $\Gamma \vdash (x :_U A) \to B : s$, then there exists sort t and natural number i such that $\Gamma \vdash A : U_i$ and $\Gamma, x :_U A \vdash B : t_i$.*

Proof. By induction on the derivation of $\Gamma \vdash (x :_U A) \to B : s$. □

Lemma G.2. *For any context Γ and terms A, B, s, if there is $\Gamma \vdash (x :_L A) \to B : s$, then there exists sort t and natural number i such that $\Gamma \vdash A : L_i$ and $\Gamma \vdash B : t_i$.*

Proof. By induction on the derivation of $\Gamma \vdash (x :_L A) \to B : s$. □

Lemma G.3. *For any context Γ and terms A, B, s, if there is $\Gamma \vdash (x :_U A) \multimap B : s$, then there exists sort t and natural number i such that $\Gamma \vdash A : U_i$ and $\Gamma, x :_U A \vdash B : t_i$.*

Proof. By induction on the derivation of $\Gamma \vdash (x :_U A) \multimap B : s$. □

Lemma G.4. *For any context Γ and terms A, B, s, if there is $\Gamma \vdash (x :_L A) \multimap B : s$, then there exists sort t and natural number i such that $\Gamma \vdash A : L_i$ and $\Gamma \vdash B : t_i$.*

Proof. By induction on the derivation of $\Gamma \vdash (x :_L A) \multimap B : s$. □

Lemma G.5. *For any context Γ, terms A, n, C and sort s, if there is $\Gamma \vdash \lambda x :_s A.n : C$, then for all terms A', B, sorts s', t and natural number i such that $C \preceq (x :'_s A') \to B$ and $\overline{\Gamma, x :'_s A'} \vdash B : t_i$, there is $\Gamma, x :'_s A' \vdash n : B$.*

Proof. By induction on the derivation of $\Gamma \vdash \lambda x :_s A.n : C$ and Lemmas G.1, G.2. □

Lemma G.6. *For any context Γ, terms A, n, C and sort s, if there is $\Gamma \vdash \lambda x :_s A.n : C$, then for all terms A', B, sorts s', t and natural number i such that $C \preceq (x :'_s A') \multimap B$ and $\overline{\Gamma, x :'_s A'} \vdash B : t_i$, there is $\Gamma, x :'_s A' \vdash n : B$.*

Proof. By induction on the derivation of $\Gamma \vdash \lambda x :_s A.n : C$ and Lemmas G.3, G.4. □

Lemma G.7. *For any context Γ, terms A, A', B, n, sorts s, s', t and natural number i, if there is $\overline{\Gamma} \vdash (x :'_s A') \to B : t_i$ and $\Gamma \vdash \lambda x :_s A.n : (x :'_s A') \to B$, then there is $\Gamma, x :'_s A' \vdash n : B$.*

Proof. Direct consequence of Lemmas G.1, G.2 and G.5. □

Lemma G.8. *For any context Γ, terms A, A', B, n, sorts s, s', t and natural number i, if there is $\overline{\Gamma} \vdash (x :'_s A') \multimap B : t_i$ and $\Gamma \vdash \lambda x :_s A.n : (x :'_s A') \multimap B$, then there is $\Gamma, x :'_s A' \vdash n : B$.*

Proof. Direct consequence of Lemmas G.3, G.4 and G.6. □

H Weakening Lemmas of CLC

Weakening for non-linear types is admissible in CLC. To prove this, we first define an *agreeR* relation between two contexts Γ, Γ' and a mapping ξ from variables to variables.

$$\frac{}{agreeR\ \xi\ \epsilon\ \epsilon}\ \text{AGREER-}\epsilon$$

$$\frac{agreeR\ \xi\ \Gamma\ \Gamma' \qquad x \notin FV(\Gamma) \cup FV(\Gamma')}{agreeR\ (\xi \cup (x,x))\ (\Gamma, x :_U A)(\Gamma', x :_U A[\xi])}\ \text{AGREER-U}$$

$$\frac{agreeR\ \xi\ \Gamma\ \Gamma' \qquad x \notin FV(\Gamma) \cup FV(\Gamma')}{agreeR\ (\xi \cup (x,x))\ (\Gamma, x :_L A)(\Gamma', x :_L A[\xi])}\ \text{AGREER-L}$$

$$\frac{agreeR\ \xi\ \Gamma\ \Gamma' \qquad x \notin FV(\Gamma) \cup FV(\Gamma')}{agreeR\ \xi\ \Gamma\ (\Gamma', x :_U A)}\ \text{AGREER-WK}$$

H.1 Properties of agreeR

Lemma H.1. *For any context Γ and the identity map id from variables to variables, agreeR id Γ Γ is always true.*

Proof. By induction on the structure of Γ and the definition of *agreeR*. □

Lemma H.2. *For contexts Γ, Γ' and mapping ξ, if there is agreeR ξ Γ Γ' and $|\Gamma|$, then there is $|\Gamma'|$.*

Proof. By induction on the derivation of *agreeR* ξ Γ Γ'. □

Lemma H.3. *For contexts Γ, Γ' and mapping ξ, if there is agreeR ξ Γ Γ', then there is agreeR ξ $|\Gamma|$ $|\Gamma'|$.*

Proof. By induction on the derivation of *agreeR* ξ Γ Γ'. □

H.2 Weakening Theorem

Lemma H.4. *For contexts $\Gamma, \Gamma', \Gamma_1, \Gamma_2$ and mapping ξ, if there is agreeR ξ Γ Γ' and merge Γ_1 Γ_2 Γ, then there exists Γ'_1, Γ'_2 such that merge Γ'_1 Γ'_2 Γ', and agreeR ξ Γ_1 Γ'_1 and agreeR ξ Γ_2 Γ'_2.*

Proof. By induction on the derivation of *agreeR* ξ Γ Γ' and lemmas in Section H.1. □

Lemma H.5. *For context Γ, Γ', terms m, A and mapping ξ, if there is $\Gamma \vdash m : A$ and agreeR ξ Γ Γ', then there is $\Gamma' \vdash m[\xi] : A[\xi]$.*

Proof. By induction on the derivation of $\Gamma \vdash m : A$. We shall only discuss the application case in detail, as the other cases are proven by application of the induction hypothesis and the lemmas in Section H.1.

- For the App-U→ case, Lemma H.4 is applied to split the context Γ into two contexts Γ'_1 and Γ'_2 such that there is *merge* Γ'_1 Γ'_2 Γ' and *agreeR* ξ Γ_1 Γ'_1 and *agreeR* ξ Γ_2 Γ'_2. From $|\Gamma_2|$ and Lemma H.2 we know that there is $|\Gamma'_2|$. At this point, the induction hypothesis allows us to apply App-U→ to prove the goal.
- For the App-L→ case, Lemma H.4 is applied to split the context Γ into two contexts Γ'_1 and Γ'_2 such that there is *merge* Γ'_1 Γ'_2 Γ' and *agreeR* ξ Γ_1 Γ'_1 and *agreeR* ξ Γ_2 Γ'_2. At this point, the induction hypothesis allows us to apply App-L→ to prove the goal.

- For the App-U–⊸ case, Lemma H.4 is applied to split the context Γ into two contexts Γ_1' and Γ_2' such that there is *merge* $\Gamma_1'\ \Gamma_2'\ \Gamma'$ and *agreeR* $\xi\ \Gamma_1\ \Gamma_1'$ and *agreeR* $\xi\ \Gamma_2\ \Gamma_2'$. From $|\Gamma_2|$ and Lemma H.2 we know that there is $|\Gamma_2'|$. At this point, the induction hypothesis allows us to apply App-U–⊸ to prove the goal.
- For the App-L–⊸ case, Lemma H.4 is applied to split the context Γ into two contexts Γ_1' and Γ_2' such that there is *merge* $\Gamma_1'\ \Gamma_2'\ \Gamma'$ and *agreeR* $\xi\ \Gamma_1\ \Gamma_1'$ and *agreeR* $\xi\ \Gamma_2\ \Gamma_2'$. At this point, the induction hypothesis allows us to apply App-L–⊸ to prove the goal.

□

Theorem H.6. *Weakening is admissible for CLC variables of non-linear type. For context Γ and terms m, A, B, if there is $\Gamma \vdash m : A$, then there is $\Gamma, x :_U B \vdash m : A$.*

Proof. Using agreeR-wk and Lemma H.1 a proof of *agreeR id* $\Gamma\ (\Gamma, x :_U B)$ can be constructed. Then by Lemma H.5, the theorem can be proven. □

I Substitution Lemmas of CLC

Similar to the proof of weakening, we first define an *agreeS* relation between two contexts Γ, Δ and a mapping σ from variables to terms.

$$\frac{}{agreeS\ \sigma\ \epsilon\ \epsilon}\ \text{agreeS-}\epsilon$$

$$\frac{agreeS\ \sigma\ \Delta\ \Gamma \qquad x \notin FV(\Delta) \cup FV(\Gamma)}{agreeS\ (\sigma \cup (x, x))\ (\Delta, x :_U A[\sigma])\ (\Gamma, x :_U A)}\ \text{agreeS-U}$$

$$\frac{agreeS\ \sigma\ \Delta\ \Gamma \qquad x \notin FV(\Delta) \cup FV(\Gamma)}{agreeS\ (\sigma \cup (x, x))\ (\Delta, x :_L A[\sigma])\ (\Gamma, x :_L A)}\ \text{agreeS-L}$$

$$\frac{agreeS\ \sigma\ \Delta\ \Gamma \qquad \overline{\Delta} \vdash n : A[\sigma] \qquad x \notin FV(\Delta) \cup FV(\Gamma)}{agreeS\ (\sigma \cup (x, n))\ \Delta\ (\Gamma, x :_U A)}\ \text{agreeS-wkU}$$

$$\frac{agreeS\ \sigma\ \Delta_1\ \Gamma \qquad merge\ \Delta_1\ \Delta_2\ \Delta \qquad \Delta_2 \vdash n : A[\sigma] \qquad x \notin FV(\Delta) \cup FV(\Gamma)}{agreeS\ (\sigma \cup (x, n))\ \Delta\ (\Gamma, x :_L A)}\ \text{agreeS-wkL}$$

$$\frac{A \preceq B \qquad \overline{\Delta} \vdash B[\sigma] : U_i \qquad agreeS\ \sigma\ \Delta\ (\Gamma, x :_U A)}{agreeS\ \sigma\ \Delta\ (\Gamma, x :_U B)}\ \text{agreeS-convU}$$

$$\frac{A \preceq B \qquad \overline{\Delta} \vdash B[\sigma] : L_i \qquad \overline{\Gamma} \vdash B : L_i \qquad agreeS\ \sigma\ \Delta\ (\Gamma, x :_L A)}{agreeS\ \sigma\ \Delta\ (\Gamma, x :_L B)}\ \text{agreeS-convL}$$

I.1 Properties of agreeS

Lemma I.1. *For any context Γ and identity mapping id, there is agreeS id Γ Γ.*

Proof. By induction on the structure of Γ. □

Lemma I.2. *For contexts Δ, Γ and mapping σ, if there is agreeS σ Δ Γ, then there is agreeS σ $\overline{\Delta}$ $\overline{\Gamma}$.*

Proof. By induction on the derivation of *agreeS* σ Δ Γ. □

I.2 Substitution Lemma

Lemma I.3. *For contexts $\Delta, \Gamma, \Gamma_1, \Gamma_2$ and mapping σ, if there is agreeS σ Δ Γ and merge Γ_1 Γ_2 Γ, then there exists contexts Δ_1, Δ_2 such that merge Δ_1 Δ_2 Δ and agreeS σ Δ_1 Γ_1 and agreeS σ Δ_2 Γ_2.*

Proof. By induction on the derivation of *agreeS* σ Δ Γ and lemmas in Section I.1. □

Lemma I.4. *Generalized Substitution Lemma. For context Γ, Δ, terms m, A and mapping σ, if there is $\Gamma \vdash m : A$ and agreeS σ Δ Γ, then there is $\Delta \vdash m[\sigma] : A[\sigma]$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash m : A$. Similar to the proof of Lemma H.5, the interesting cases are the application cases where Lemma I.3 must be utilized to split the *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ judgments for use in the induction hypothesis. □

I.3 Corollaries of Substitution

Corollary I.5. *For contexts $\Gamma_1, \Gamma_2, \Gamma$ and terms A, B, m, n, if there is $\Gamma_1, x :_U A \vdash m : B$ and $|\Gamma_2|$ and merge $\Gamma_1\ \Gamma_2\ \Gamma$ and $\Gamma_2 \vdash n : A$, then there is $\Gamma \vdash m[n/x] : B[n/x]$.*

Corollary I.6. *For contexts $\Gamma_1, \Gamma_2, \Gamma$ and terms A, B, m, n, if there is $\Gamma_1, x :_L A \vdash m : B$ and merge $\Gamma_1\ \Gamma_2\ \Gamma$ and $\Gamma_2 \vdash n : A$, then there is $\Gamma \vdash m[n/x] : B[n/x]$.*

Corollary I.7. *For context Γ, terms m, A, B, C and natural number i, if there is $B \equiv A$ and $\overline{\Gamma} \vdash A : U_i$ and $\Gamma, x :_U A \vdash m : C$, then there is $\Gamma, x :_U B \vdash m : C$.*

Corollary I.8. *For context Γ, terms m, A, B, C and natural number i, if there is $B \equiv A$ and $\overline{\Gamma} \vdash A : L_i$ and $\Gamma, x :_L A \vdash m : C$, then there is $\Gamma, x :_L B \vdash m : C$.*

J Typing Validity of CLC

In this section, we prove that the types of all CLC terms are themselves well-sorted.

Lemma J.1. *For contexts $\Gamma_1, \Gamma_2, \Gamma$, if there is merge $\Gamma_1\ \Gamma_2\ \Gamma$ and $\Gamma \vdash$, then there is $\Gamma_1 \vdash$ and $\Gamma_2 \vdash$.*

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and the properties of *merge* discussed in Section D.1. □

Theorem J.2. *The validity of typing theorem. For any context Γ and terms m, A, if there is $\Gamma \vdash$ and $\Gamma \vdash m : A$, then there exists sort s and natural number i such that $\overline{\Gamma} \vdash A : s_i$.*

K Subject Reduction of CLC

Theorem K.1. *For any context Γ and terms m, n, A, if $\Gamma \vdash$ and $\Gamma \vdash m : A$ and $m \rightsquigarrow n$, then there is $\Gamma \vdash n : A$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash m : A$. The interesting cases are the application cases which we shall discuss in detail.

- For the App-U$\to$ case, from assumptions *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and $|\Gamma_2|$ and Lemmas D.7, D.10, we can conclude that $\overline{\Gamma_1} = \overline{\Gamma}$ and $\overline{\Gamma_2} = \overline{\Gamma}$. Applying Lemma J.1 to assumptions *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and $\Gamma \vdash$ obtains $\Gamma_1 \vdash$ and $\Gamma_2 \vdash$. Now by the induction hypothesis, we can conclude there exists sorts s, t and natural numbers i, j such that there are $\overline{\Gamma_1} \vdash (x :_U A) \to B : s_i$ and $\overline{\Gamma_2} \vdash A : t_j$. Applying Lemma G.1 to assumption $\overline{\Gamma_1} \vdash (x :_U A) \to B : s_i$ allows us to derive $\overline{\Gamma_1} \vdash A : U_{i'}$ and $\overline{\Gamma_1}, x :_U A \vdash B : s'_{j'}$, where s' is a sort and i', j' are natural numbers. The goal can finally be proven by applying the substitution Lemma I.5 on assumptions $\Gamma_2 \vdash n : A$ and $\overline{\Gamma_1}, x :_U A \vdash B : s'_{j'}$.
- For the App-L$\to$ case, from assumption *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and Lemmas D.10, we can conclude that $\overline{\Gamma_1} = \overline{\Gamma}$ and $\overline{\Gamma_2} = \overline{\Gamma}$. Applying Lemma J.1 to assumptions *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and $\Gamma \vdash$ obtains $\Gamma_1 \vdash$ and $\Gamma_2 \vdash$. Now by the induction hypothesis, we can conclude that there exists sorts s, t and natural numbers i, j such that there are $\overline{\Gamma_1} \vdash (x :_L A) \to B : s_i$ and $\overline{\Gamma_2} \vdash A : t_j$. Applying Lemma G.2 to assumption $\overline{\Gamma_1} \vdash (x :_L A) \to B : s_i$ allows us to derive $\overline{\Gamma_1} \vdash A : L_{i'}$ and $\overline{\Gamma_1} \vdash B : s'_{j'}$. Due to the fact that variable x is not a free variable in B, the substitution occurring in goal $\exists s \in sort, i \in \mathbb{N}, \overline{\Gamma} \vdash B[n/x] : s_i$ is trivial, thus the judgment $\overline{\Gamma_1} \vdash B : s'_{j'}$ that we have proven shows the existence of the goal.
- For the App-U$\multimap$ case, the proof is similar to the App-U$\to$ case, the only difference is that the inversion lemmas used correspond to $\multimap$ instead of $\to$.
- For the App-L$\multimap$ case, the proof is similar to the App-L$\to$ case, the only difference is that the inversion lemmas used correspond to $\multimap$ instead of $\to$.

□

L Linearity Theorems of CLC

L.1 Linearity

We introduce a meta-function *occurs* that counts the number of times a given variable occurs in a term.

$$occurs(x, y) = \begin{cases} 1 & x =_\alpha y \\ 0 & x \neq_\alpha y \end{cases}$$
$$occurs(x, s_i) = 0$$
$$occurs(x, ((y :_s A) \to B)) = occurs(x, A) + occurs(x, B)$$
$$occurs(x, ((y :_s A) \multimap B)) = occurs(x, A) + occurs(x, B)$$
$$occurs(x, (\lambda x :_s A.n)) = occurs(x, A) + occurs(x, n)$$
$$occurs(x, (m\ n)) = occurs(x, m) + occurs(x, n)$$

Lemma L.1. *For contexts $\Gamma_1, \Gamma_2, \Gamma$, if there is merge $\Gamma_1\ \Gamma_2\ \Gamma$, then for any variable with linear type $x \in \Gamma$ there is $x \in \Gamma_1$ and $x \notin \Gamma_2$ or $x \in \Gamma_2$ and $x \notin \Gamma_1$.*

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma L.2. *For contexts $\Gamma_1, \Gamma_2, \Gamma$, if there is merge $\Gamma_1\ \Gamma_2\ \Gamma$, then for any variable $x \notin \Gamma$ there is $x \notin \Gamma_1$ and $x \notin \Gamma_2$.*

Proof. By induction on the derivation of *merge* $\Gamma_1\ \Gamma_2\ \Gamma$. □

Lemma L.3. *For context Γ, terms m, A, if there is $\Gamma \vdash m : A$, then for any variable $x \notin \Gamma$ there is $occurs(x, m) = 0$*

Proof. By induction on the derivation of $\Gamma \vdash m : A$. □

Theorem L.4. *Linearity. For context Γ, terms m, A, if there is $\Gamma \vdash m : A$, then for any variable with linear type $x \in \Gamma$ there is $occurs(x, m) = 1$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash m : A$, we will discuss the application cases in detail.

- For case App-U$\to$, by assumptions *merge* $\Gamma_1\ \Gamma_2\ \Gamma$ and $(x :_L A) \in \Gamma$ and Lemma L.1 we can conclude that $x \in \Gamma_1$ and $x \notin \Gamma_2$ or $x \in \Gamma_2$ and $x \notin \Gamma_1$. In both cases, applying the induction hypothesis and Lemma L.3 proves the goal.
- For case App-L$\to$, the proof is the same as App-U$\to$.
- For case App-U$\multimap$, the proof is the same as App-U$\to$.
- For case App-L$\multimap$, the proof is the same as App-U$\to$.

□

L.2 Promotion

Theorem L.5. *Promotion. For context Γ, terms m, A, B and sort s, if there is $|\Gamma|$ and $\Gamma \vdash$ and $\Gamma \vdash m : (x :_s A) \multimap B$, then there exists term n such that $\Gamma \vdash n : (x :_s A) \to B$.*

Proof. Set $n = \lambda x :_s A.(m\ x)$. The proof proceeds by case analysis on the sort s.

- If $s = U$, then we may apply Theorem J.2 to assumption $\Gamma \vdash m : (x :_U A) \multimap B$ to show that there exists sort t and natural number i such that there is $\overline{\Gamma} \vdash (x :_U A) \multimap B : t_i$. Now applying Lemma G.3 to $\overline{\Gamma} \vdash (x :_U A) \multimap B : t_i$ shows that there exists sort t' and natural number i' such that $\overline{\Gamma} \vdash A : U_{i'}$ and $\overline{\Gamma}, x :_U A \vdash B : t'_{i'}$. Now by U$\to$ and Lemma D.13 the goal is proven.

- If $s = L$, then we may apply Theorem J.2 to assumption $\Gamma \vdash m : (x :_L A) \multimap B$ to show that there exists sort t and natural number i such that $\overline{\Gamma} \vdash (x :_L A) \multimap B : t_i$. Now applying Lemma G.4 to $\overline{\Gamma} \vdash (x :_L A) \multimap B : t_i$ shows that there exists sort t' and natural number i' such that $\overline{\Gamma} \vdash A : U_{i'}$ and $\overline{\Gamma} \vdash B : t'_{i'}$. Now by L$\to$ and Lemma D.13 the goal is proven.

□

L.3 Dereliction

Theorem L.6. *Dereliction. For context Γ, terms m, A, B and sort s, if there is $\Gamma \vdash$ and $\Gamma \vdash m : (x :_s A) \to B$, then there exists term n such that $\Gamma \vdash n : (x :_s A) \multimap B$.*

Proof. Set $n = \lambda x :_s A.(m\ x)$. The proof proceeds by case analysis on the sort s.

- If $s = U$, then we may apply Theorem J.2 to $\Gamma \vdash m : (x :_U A) \to B$ showing that there exists sort t and natural number i such that there is $\overline{\Gamma} \vdash (x :_U A) \to B : t_i$. Now applying Lemma G.1 to $\overline{\Gamma} \vdash (x :_U A) \to B : t_i$ shows that there exists sort t' and natural number i' such that $\overline{\Gamma} \vdash A : U_{i'}$ and $\overline{\Gamma}, x :_U A \vdash B : t'_{i'}$. By U$\multimap$ and Lemma D.14 we can prove $\overline{\Gamma} \vdash (x :_U A) \multimap B : L_{i'}$. By rule $\lambda\multimap$, the rest of the goal can be proven in a straightforward manner.
- If $s = L$, then we may apply Theorem J.2 to $\Gamma \vdash m : (x :_L A) \to B$ showing that there exists sort t and natural number i such that there is $\overline{\Gamma} \vdash (x :_L A) \to B : t_i$. Now applying Lemma G.2 to $\overline{\Gamma} \vdash (x :_L A) \to B : t_i$ shows that there exists sort t' and natural number i' such that $\overline{\Gamma} \vdash A : U_{i'}$ and $\overline{\Gamma} \vdash B : t'_{i'}$. By L$\multimap$ and Lemma D.14 we can prove $\overline{\Gamma} \vdash (x :_L A) \multimap B : L_{i'}$. By rule $\lambda\multimap$, the rest of the goal can be proven in a straightforward manner.

□

M Logical Consistency of CLC

M.1 Strong Normalization

The proof of the logical consistency of CLC proceeds by construction of a reduction preserving erasure from CLC to CCω. As CCω is consistent, CLC must be consistent as well.

The erasure procedure is recursively defined as follows.

$$\begin{aligned}
[\![x]\!] &= x \\
[\![U_i]\!] &= Type_i \\
[\![L_i]\!] &= Type_i \\
[\![(x :_s A) \to B]\!] &= (x : [\![A]\!]) \to [\![B]\!] \\
[\![(x :_s A) \multimap B]\!] &= (x : [\![A]\!]) \to [\![B]\!] \\
[\![\lambda x :_s A.n]\!] &= \lambda x : [\![A]\!].[\![n]\!] \\
[\![m\ n]\!] &= [\![m]\!]\ [\![n]\!]
\end{aligned}$$

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

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

Lemma M.1. *For CLC term m, map σ from variables to CLC terms, map τ from variables to CCω terms, if for all variables x there is $[\![\sigma\ x]\!] = \tau\ x$, then $[\![m[\sigma]]\!] = [\![m]\!][\tau]$.*

Proof. By induction on the structure of term m. □

For the following lemmas, we will index relations and judgments with subscript CLC or CCω to emphasize the language it is defined over.

Lemma M.2. *For any CLC terms m and n, if there is $m \rightsquigarrow_{CLC} n$, then there is $[\![m]\!] \rightsquigarrow_{CC\omega} [\![n]\!]$.*

Proof. By induction on the derivation of $m \rightsquigarrow_{\text{CLC}} n$. □

Lemma M.3. *For any CLC terms m and n, if there is $m \equiv_{CLC} n$, then there is $[\![m]\!] \equiv_{CC\omega} [\![n]\!]$.*

Proof. By induction on the derivation of $m \equiv_{\text{CLC}} n$ and Lemma M.2. □

Lemma M.4. *For any CLC terms m and n, if there is $m \prec_{CLC} n$, then there is $[\![m]\!] \prec_{CC\omega} [\![n]\!]$.*

Proof. By induction on the derivation of $m \prec_{\text{CLC}} n$. □

Lemma M.5. *For any CLC terms m and n, if there is $m \preceq_{CLC} n$, then there is $[\![m]\!] \preceq_{CC\omega} [\![n]\!]$.*

Proof. By case analysis on the derivation of $m \preceq_{\text{CLC}} n$ and the properties of subtyping proven in Section E. □

Theorem M.6. *Embedding. For any CLC context Γ and CLC terms m, A, if there is $\Gamma \vdash_{CLC} m : A$, then there is $[\![\Gamma]\!] \vdash_{CC\omega} [\![m]\!] : [\![A]\!]$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\text{CLC}} m : A$. □

Corollary M.7. *For any CLC context Γ, if there is $\Gamma \vdash_{CLC}$, then there is $[\![\Gamma]\!] \vdash_{CC\omega}$.*

Proof. Direct consequence of applying Theorem M.6 to all types in context Γ. □

Theorem M.8. *Strong normalization of CLC.*

Proof. Suppose there exists a well-typed CLC term m with an infinite sequence of reductions. Theorem M.6 shows that there must exist some term $[\![m]\!]$ that is well-typed in CCω. Additionally, this infinite sequence of reductions on m can be translated in CCω step-wise by Lemma M.2. This shows that we have constructed a non-normalizing CCω term $[\![m]\!]$, which a contradiction to the strong normalization property of CCω, thus CLC must be strongly normalizing as well. □

M.2 Embedding

To show that CLC is compatible with the predicative fragment of CCω, we construct a lifting procedure that lifts CCω terms into CLC in a straightforward way.

$$\begin{aligned}
(\!|x|\!) &= x \\
(\!|Type_i|\!) &= U_i \\
(\!|(x : A) \to B|\!) &= (x :_U (\!|A|\!)) \to (\!|B|\!) \\
(\!|\lambda x : A.n|\!) &= \lambda x :_U (\!|A|\!).(\!|n|\!) \\
(\!|m\ n|\!) &= (\!|m|\!)\ (\!|n|\!)
\end{aligned}$$

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

$$\begin{aligned}
(\!|\epsilon|\!) &= \epsilon \\
(\!|\Gamma, x : A|\!) &= (\!|\Gamma|\!), x :_U (\!|A|\!)
\end{aligned}$$

Lemma M.9. *For CCω context Γ, there is $|(\!|\Gamma|\!)|$.*

Proof. By induction on the structure of Γ. □

Lemma M.10. *For CCω term m, map σ from variables to CCω terms, map τ from variable to CLC terms, if for all variables x there is $(\!|\sigma\ x|\!) = \tau\ x$, then $(\!|m[\sigma]|\!) = (\!|m|\!)[\tau]$.*

Proof. By induction on the structure of term m. □

For the following lemmas, we will index relations and judgments with subscript CLC of CCω to emphasize the language it is defined over.

Lemma M.11. *For any CCω terms m and n, if there is $m \rightsquigarrow_{CC\omega} n$, then there is $(\!|m|\!) \rightsquigarrow_{CLC} (\!|n|\!)$.*

Proof. By induction on the derivation of $m \rightsquigarrow_{CC\omega} n$. □

Lemma M.12. *For any CCω terms m and n, if there is $m \equiv_{CC\omega} n$, then there is $(\!|m|\!) \equiv_{CLC} (\!|n|\!)$.*

Proof. By induction on the derivation of $m \equiv_{CC\omega} n$ and Lemma M.11. □

Lemma M.13. *For any CCω terms m and n, if there is $m \prec_{CC\omega} n$, then there is $(\!|m|\!) \prec_{CLC} (\!|n|\!)$.*

Proof. By induction on the derivation of $m \prec_{CC\omega} n$. □

Lemma M.14. *For any CCω terms m and n, if there is $m \preceq_{CC\omega} n$, then there is $(\!|m|\!) \preceq_{CLC} (\!|n|\!)$.*

Proof. By case analysis on the derivation of $m \preceq_{CC\omega} n$ and the properties of subtyping proven in Section E. □

Theorem M.15. *Lifting. For any CCω context Γ and CCω terms m, A, if there is $\Gamma \vdash_{CC\omega} m : A$, then there is $(\!|\Gamma|\!) \vdash_{CLC} (\!|m|\!) : (\!|A|\!)$.*

Proof. By induction on the derivation of $\Gamma \vdash_{CC\omega} m : A$. □