The Calculus of Linear Constructions

Qiancheng Fu

October 10, 2021

Abstract

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

1 Introduction

The Calculus of Constructions (CC) is a dependent type theory introduced by Coquand, and Huet in their landmark work [11]. In CC types can depend on terms, allowing one to write precise specifications as types. Today, CC and its variations CIC [27] and ECC [26] lie at the core of popular proof assistants such as Coq [30], Agda [25], Lean [12], and others. These theorem provers have found great success in the fields of software verification [19, 2], and constructive mathematics [16, 7].

However, due to its origins as a logical framework for constructive mathematics, it is quite difficult for CC to encode and reason about resources. Intuitively, a mathematical theorem can be applied an unrestricted number of times. Comparatively, the usage of resources is more limited. For example, if we encode Girard's classical example [15] of purchasing cigarettes literally into CC as a function of type:

$$Money \to Camels + Marlboro$$

If viewed as an propositional implication, the customer will still maintain full ownership of their money after paying the vendor, because implication does not diminish the validity of its antecedent. Unless the vendor is exceedingly generous, we are faced with the crime of counterfeiting. Users of proof assistants based on CC often need to embed external logics [9] to provide additional reasoning principles for dealing with resource. The design and embedding of these logics is a difficult problem in its own right, requiring additional proofs to justify its soundness. We propose an alternative solution: extend CC with linear types.

Linear Logic is a substructural logic introduced by Girard in his seminal work [14]. Girard notice that the weakening and contraction rules of Classical Logic when restricted carefully, gives rise to a new logical foundation for reasoning about resource. Wadler [32, 33] first notice that an analogous restriction to variable usage in simple type theory leads to a linear type theory, where terms respect resources. A term calculus for linear type theory was later realized by Abramsky [1]. Benton [4] investigates the ramifications of the ! exponential in linear term calculi, decomposing it to adjoint connectives F and G that map between linear and non-linear judgments. Programming languages [23, 35, 5] featuring linear types have also been implemented, allowing programmers to write resource safe software in practical applications. The success of integrating Linear Logic with simple type theory exposes a tantalizing new frontier of integrating linearity with richer type theories.

Work have been done to extend dependent type theories with linear types. Cervesato and Pfenning extends the Edinburgh Logical Framework with linear types [17, 8], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár [31] gives a categorical semantics for linear dependent types. Krishnaswami et al. present a dependent linear type theory [18] based on Benton's early work of mixed linear and non-linear calculus, demonstrating the ability to internalize imperative programming the style of Hoare Type Theory [24]. Luo et al. introduce the property of essential linearity, and a mixed linear/non-linear context, describing the first type theory that allows types to depend on linear terms. Based on initial ideas of McBride [22], Atkey's Quantitative Type Theory (QTT) [3] uses semi-ring annotations to track variable occurrence, simulating irrelevance, linear, and affine types within a unified framework. The Idris 2 programming language [6] implements QTT as its core type system.

We propose a new linear dependent type system - The Calculus of Linear Constructions (CLC). CLC extends CCω with linear types. CCω itself is an extension of CC with a cumulative hierarchy of type universes. We add extra universes L of linear types with cumulativity parallel to the universes U of non-linear types. Universe information is propagated by an indexed typing judgment down to the term level, controlling the usage of weakening and contraction rules. This ultimately results in the *linearity* theorem, stating that all resources are used exactly once.

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

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

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

Compared to preexisting approaches for integrating linear types and dependent types, CLC offers a "lightweight" approach to extending CCω with linear types, akin to Mazurak et al.'s work on System F [21]. This allows for a straightforward modeling of CLC in CCω, and lifting of CCω into CLC, endowing CLC with the fruits of decades of CC research. We further extend CLC with inductive types, showing that as a programming language it can manipulate mutable data structures in a principled way. We have formalized all major results in Coq, and implemented a prototype in OCaml.

Contributions: Our contributions can be summarized as follows.

- Fist, we describe the Calculus of Linear Constructions, an extension to the Calculus of Constructions with linear types. The integration of linear types and dependent types allows CLC to directly and precisely reason about resource.
- Next, we study the meta-theory of CLC directly, showing that it satisfies the standard properties of confluence, regularity, and subject reduction.
- We observe that CLC is highly backwards compatible with CCω. We construct a reduction preserving model of CLC in CCω, showing that CLC is consistent.
- All major results have been formalized and proven correct in the Coq Proof Assistant with help from the Autosubst [29] library. To the best of our knowledge, our development is the first machine checked formalization of a linear dependent type theory.
- Furthermore, we extend CLC with linear inductive data, demonstrating that as a programming language, CLC can safely manipulate mutable data structures.
- Finally, we give an implementation extended with user definable linear and non-linear inductive types. Algorithmic type checking employed by the implementation streamlines the process of writing CLC.

2 The Language of CLC

2.1 Syntax

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

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

Figure 1: Syntax

A clear departure of our language from standard presentations of both linear type theory, and dependent type theory is the indexed function type: $(x : A)\, {}_s\!\to_t B$. The reason for these indices is that we have built the ! exponential of linear logic directly into universe sorts. The indices s, and t annotate the domain, and co-domain's respective universe sort, hence annotating linearity. The behavior of ! is difficult to account for even in simple linear type theory. Subtle issues arise if !! is not canonically isomorphic to !, which may invalidate the substitution lemma [34]. By integrating the exponential directly into type formation rules, we implicitly limit ! to only be used canonically. This allows us to derive the substitution lemma, and construct a direct modeling of CLC in CCω without needing extra machinery for manipulating exponential.

$$(\_ : A)\, {}_U\!\to_U B \quad \equiv \quad !A \multimap !B \tag{1}$$
$$(\_ : A)\, {}_U\!\to_L B \quad \equiv \quad !A \multimap B \tag{2}$$
$$(\_ : A)\, {}_L\!\to_U B \quad \equiv \quad A \multimap !B \tag{3}$$
$$(\_ : A)\, {}_L\!\to_L B \quad \equiv \quad A \multimap B \tag{4}$$

Figure 2: Correspondence of CLC types and MELL implications

Figure 2 illustrates the correspondence between CLC function types and Multiplicative Exponential Linear Logic (MELL) implications. MELL lacks counterparts for the cases (1), (2) if the co-domain B is dependent on arguments of the domain A. Though it may seem as if implications of the $!(A \multimap B)$ form are left out, CLC encodes these by endowing each function type with a universe sort in their formation judgment. The encoding for this example would be $\Gamma \vdash A\, {}_L\!\to_L B :_U U_i$. We will discuss function type formation in Section 2.5 in more detail.

2.2 Universes and Cumulativity

CLC features two sorts of universes U, and L with level indices $*, 0, 1, 2, \cdots$. U_* is the impredicative universe of propositions. U_k, and L_k are the predicative universe of non-linear types, and linear types

respectively. The main mechanism that CLC uses to distinguish between linear and non-linear types is by checking the universe to which they belong. Basically, terms with types that occur within U_i are unrestricted in their usage. Terms with types that occur within L_k are restricted to being used exactly once.

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

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

1. $\preceq$ is a partial order with respect to definitional equality.
   (a) If $A \equiv B$, then $A \preceq B$.
   (b) If $A \preceq B$ and $B \preceq A$, then $A \equiv B$.
   (c) If $A \preceq B$ and $B \preceq C$, then $A \preceq B$.
2. $U_* \preceq U_0 \preceq U_1 \preceq U_2 \preceq \cdots$
3. $L_0 \preceq L_1 \preceq L_2 \preceq \cdots$
4. If $A_1 \equiv A_2$ and $B_1 \preceq B_2$, then $(x : A_1) \to_{s\,t} B_1 \preceq (x : A_2) \to_{s\,t} B_2$

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

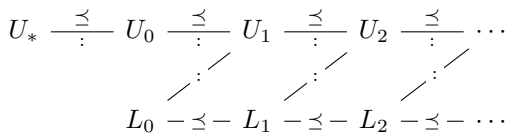

Figure 3: The Universe Hierarchy

2.3 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[20]. Variables in the context are annotated to indicate whether they are linear or non-linear. A non-linear variable is annotated as $\Gamma, x :_U A$, whereas a linear variable is annotated as $\Gamma, x :_L A$.

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

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

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

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

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

Figure 4: Structural Judgments

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

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

2.4 Typing Judgment

Typing judgments in CLC take on the form of $\Gamma \vdash m :_s A$. Intuitively, this judgment states that the term m is an inhabitant of type A, with free variables typed in Γ. The sort index s tells us the linearity of m. Specifically, if $s = U$, then m has unrestricted usage. Conversely, if $s = L$, then m must be used exactly once. In Section 2.8, we show through the regularity theorem that s corresponds exactly to the sort of A's universe.

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

1. A *type* A is *non-linear* under context Γ if $\Gamma \vdash A :_U U_i$.
2. A *type* A is *linear* under context Γ if $\Gamma \vdash A :_U L_k$.
3. A *term* m is *unrestricted* under context Γ if it has type A, and $\Gamma \vdash m :_U A$.
4. A *term* m is *restricted* under context Γ if it has type A, and $\Gamma \vdash m :_L A$

2.5 Type Formation

The rules for forming types are presented in Figure 5. In CLC, we forbid types from depending on linear terms similar to [8, 18] for the same reason of avoiding philosophical troubles.

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

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

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

Figure 5: Type Formation

The axiom rules Prop-Axiom, U-Axiom, L-Axiom are almost standard, the main difference being the extra side-condition of judgment $|\Gamma|$. In most presentations of dependent type theories without linear types, the universe axioms are derivable under any well-formed context Γ. Variables not pertaining to actual proofs could be introduced this way, thus giving rise to the admissibility of weakening. To support linear types, we must restrict weakening to non-linear variables. This justifies the restriction of Γ to contain only non-linear variables for Prop-Axiom, U-Axiom, L-Axiom. From L-Axiom we can see that the universe of linear types L_k is an inhabitant of U_{k+1}. This is reminiscent of Krishnaswami et al.'s treatment of linear universes [18], where linear *types* themselves can be used unrestrictedly.

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

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

The L-Prod rule is used for forming function types with linear domains. From the judgment $\Gamma \vdash A :_U L_k$,

2.6 Term Formation

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

$$\frac{|\Gamma| \qquad \Gamma \vdash (x : A) \to_s B :_U U_i \qquad \Gamma, x :_U A \vdash n :_s B}{\Gamma \vdash \lambda x.n :_U (x : A) \to_s B}\text{U-}\lambda_1$$

$$\frac{|\Gamma| \qquad \Gamma \vdash A \to_s B :_U U_k \qquad \Gamma, x :_L A \vdash n :_s B}{\Gamma \vdash \lambda x.n :_U A \to_s B}\text{U-}\lambda_2$$

$$\frac{\overline{\Gamma} \vdash (x : A) \multimap_s B :_U L_k \qquad \Gamma, x :_U A \vdash n :_s B}{\Gamma \vdash \lambda x.n :_L (x : A) \multimap_s B}\text{L-}\lambda_1$$

$$\frac{\overline{\Gamma} \vdash A \multimap_s B :_U L_k \qquad \Gamma, x :_L A \vdash n :_s B}{\Gamma \vdash \lambda x.n :_L A \multimap_s B}\text{L-}\lambda_2$$

$$\frac{\Gamma_1 \vdash m :_U (x : A) \to_s B \qquad \overline{\Gamma_2} \vdash n :_U A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B[n/x]}\text{U-App-1}$$

$$\frac{\Gamma_1 \vdash m :_U A \to_s B \qquad \Gamma_2 \vdash n :_L A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B}\text{U-App-2}$$

$$\frac{\Gamma_1 \vdash m :_L (x : A) \multimap_s B \qquad \overline{\Gamma_2} \vdash n :_U A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B[n/x]}\text{L-App-1}$$

$$\frac{\Gamma_1 \vdash m :_L A \multimap_s B \qquad \Gamma_2 \vdash n :_L A \qquad \Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma}{\Gamma \vdash m\ n :_s B}\text{L-App-2}$$

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

Figure 6: Term Formation

2.7 Reduction and Equality

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

$$\frac{n \rightsquigarrow n'}{\lambda x.n \rightsquigarrow \lambda x.n'}\text{P-}\lambda \qquad \frac{m \rightsquigarrow m' \qquad n \rightsquigarrow n'}{m\ n \rightsquigarrow m'\ n'}\textsc{P-App} \qquad \frac{m \rightsquigarrow m' \qquad n \rightsquigarrow n'}{(\lambda x.m)\ n \rightsquigarrow m'[n'/x]}\text{P-}\beta$$

$$\frac{A \rightsquigarrow A' \qquad B \rightsquigarrow B'}{(x : A) \to_s B \rightsquigarrow (x : A') \to_s B'}\textsc{P-U-Prod} \qquad \frac{A \rightsquigarrow A' \qquad B \rightsquigarrow B'}{A \to_s B \rightsquigarrow A' \to_s B'}\textsc{P-Arrow}$$

$$\frac{A \rightsquigarrow A' \qquad B \rightsquigarrow B'}{(x : A) \multimap_s B \rightsquigarrow (x : A') \multimap_s B'}\textsc{P-L-Prod} \qquad \frac{A \rightsquigarrow A' \qquad B \rightsquigarrow B'}{A \multimap_s B \rightsquigarrow A' \multimap_s B'}\textsc{P-Lolli}$$

Figure 7: Equality and Parallel Reduction

2.8 Meta Theory

We have proven the type soundness of our language in the form of *progress* and *preservation* theorems. The proofs have been formalized in Coq with help from the Autosubst[29] library.

2.8.1 Step and Parallel Step

The following lemmas and proofs are entirely standard. The restriction to value-form arguments for β-reduction does not pose any complications.

Lemma 2.1. *Single step reduction implies parallel step reduction. If $m \rightsquigarrow m'$, then $m \rightsquigarrow m'$.*

Lemma 2.2. *Parallel reduction satisfies the diamond property. If $m \rightsquigarrow m_1$ and $m \rightsquigarrow m_2$ then there exists m' such that $m_1 \rightsquigarrow m'$ and $m_2 \rightsquigarrow m'$.*

Corollary 2.2.1. *The transitive reflexive closure of parallel reduction is confluent. If $m \rightsquigarrow^* m_1$ and $m \rightsquigarrow^* m_2$ then there exists m' such that $m_1 \rightsquigarrow^* m'$ and $m_2 \rightsquigarrow^* m'$.*

Corollary 2.2.2. *The definitional equality relation is an equivalence relation.*

2.8.2 Substitution

Though the *substitution* lemma is widely considered a boring and bureaucratic theorem, it is surprisingly hard to design linear typed languages where the *substitution* lemma is admissible. Much of this difficulty arise during the substitution of non-linear expressions. Perhaps the most famous work detailing the issues of substitution is due to Wadler[34]. Since computation arise as a consequence of substitution, it is imperative to get it right.

Generally, the application rule looks similar to the following for languages with linear types.

$$\frac{\Gamma \vdash m : A \multimap B \qquad \Delta \vdash n : A}{\Gamma, \Delta \vdash m\ n : B}$$

If type A is a non-linear type, then n ought to be used freely. However, it is possible for n to contain linear variables present in Δ. If m is a lambda abstraction $\lambda x.m'$ where x occurs multiple times within m', substitution of n for x will cause duplication of linear variables. One approach for solving

this issue is to wrap non-linear values in an explicit modality, unpacking the internal value only when needed[34, 18]. Another is to ban non-linear expressions from containing linear variables[8, 20, 3].

Our call-by-value semantics resolves the substitution problem without imposing any of the modifications mentioned previously to the typing of application. The crucial realization is the following *value soundness* lemma: non-linear *values* contain no linear variables.

Lemma 2.3. *Value soundness. If* $\Gamma \vdash v :_U A$ *then* Γ.

From the *value soundness* lemma, the standard rule for application is admissible. Intuitively, a single copy of each linear resource is used to create a single non-linear value. This value can then be freely duplicated without needing the original resources to generate fresh copies. This is consistent with the common practice of retrieving non-linear values from linear references.

Lemma 2.4. *Substitution. For* $\Gamma_1, x :_s A \vdash m :_t B$ *and* $\Gamma_2 \vdash v :_s A$*, if there exists* Γ *such that merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *is defined, then* $\Gamma \vdash [v/x]m :_t [v/x]B$.

2.8.3 Type Soundness

The following theorems are a direct result of the *substitution* lemma and various canonical-form lemmas.

Theorem 2.5. *Progress. For* $\epsilon \vdash m :_s A$*, either* m *is a value or there exists* n *such that* $m \rightsquigarrow n$.

Theorem 2.6. *Preservation. For* $\Gamma \vdash m :_s A$*, if* $m \rightsquigarrow n$ *then* $\Gamma \vdash n :_s A$.

3 Extensions

3.1 Data Types

Though it is possible to encode data and propositions directly in the core language using Church-encodings, it is incredibly inconvenient. To address this, our implementation allows users to define inductive data types[13] similar to Coq or Agda[25]. A pattern matching construct[10] is defined on inductive types for data elimination. Checking is performed to ensure that non-linear inductive types do not carry linear terms and linear types cannot be used as type indices or parameters.

Commonly used inductive types are formalized in Figures 8 with their Introduction and Elimination rules formalized in Figure 9 and Figure 10 respectively. Reduction for data types obey the same call-by-value semantics as outlined in 2.7. Other standard data types that are not covered here are $\mathbb{N}$ for natural numbers and $\top$ for the unit type.

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma, x :_U A \vdash B :_U U}{\Gamma \vdash \Sigma x : A.B :_U U}\Sigma \qquad \frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma, x :_U A \vdash B :_U L}{\Gamma \vdash Fx : A.B :_U L}F$$

$$\frac{\Gamma \qquad \Gamma \vdash A :_U L \qquad \Gamma \vdash B :_U L}{\Gamma \vdash A \otimes B :_U L}\otimes \qquad \frac{\Gamma \qquad \Gamma \vdash m :_U A \qquad \Gamma \vdash n :_U A}{\Gamma \vdash m =_A n :_U U}$$

Figure 8: Data Formation

$$\frac{\Gamma_1 \vdash m :_U A \qquad \Gamma_2 \vdash n :_U (\lambda x.B)\ m \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash (m,n) :_U \Sigma x : A.B}\ \Sigma\text{-INTRO}$$

$$\frac{\Gamma_1 \vdash m :_U A \qquad \Gamma_2 \vdash n :_L (\lambda x.B)\ m \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash [m,n] :_L Fx : A.B}\ F\text{-INTRO}$$

$$\frac{\Gamma_1 \vdash m :_L A \qquad \Gamma_2 \vdash n :_L B \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \langle m,n\rangle :_L A \otimes B}\ \otimes\text{-INTRO} \qquad \frac{\Gamma \qquad \Gamma \vdash n :_U A}{\Gamma \vdash \text{refl}\ n :_U n =_A n}$$

Figure 9: Data Introduction

$$\frac{\Gamma_1 \vdash m :_U \Sigma x.A.B \qquad \Gamma_2, x :_U A, y :_U B \vdash n :_s C \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \text{let (x , y) := m in n:}_s C}\ \Sigma\text{-ELIM}$$

$$\frac{\Gamma_1 \vdash m :_L Fx.A.B \qquad \Gamma_2, x :_U A, y :_L B \vdash n :_s C \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \text{let [x , y] := m in n:}_s C}\ F\text{-ELIM}$$

$$\frac{\Gamma_1 \vdash m :_L A \otimes B \qquad \Gamma_2, x :_L A, y :_L B \vdash n :_s C \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \text{let } \langle x,y\rangle := \text{m in n:}_s C}\ \otimes\text{-ELIM}$$

$$\frac{\Gamma_1 \vdash p :_U m =_A n \qquad \Gamma_2 \vdash q :_s B[m] \qquad \overline{\Gamma}, x :_U A \vdash B[x] :_U s \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \text{subst}(\lambda x.B, p, q) :_s B[n]}$$

Figure 10: Data Elimination

3.2 Imperative Programming

With the addition of data types, axioms for imperative programming can be added to the language. In Figure 11 we give a set of axioms for stateful programs. Note that these axioms are not associated with any reductions. This unsurprisingly breaks the *Progress Theorem* as axioms are neither values nor can they reduce. Despite this obvious shortcoming, the axiomatic treatment of state provides an interface for extraction of imperative code with safe manual memory management.

3.2.1 State Programs

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma \vdash m :_U A \qquad \Gamma \vdash l :_U \mathbb{N}}{\Gamma \vdash l \mapsto_A m :_U L}\textsc{Capability} \qquad \frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma \vdash m :_U A}{\Gamma \vdash \text{new}(m) :_L Fl : \mathbb{N}.l \mapsto_A m}\textsc{New}$$

$$\frac{\Gamma \qquad \Gamma \vdash c :_L l \mapsto_A m}{\Gamma \vdash \text{free}(c) :_U \top}\textsc{Free} \qquad \frac{\Gamma \vdash c :_L l \mapsto_A m}{\Gamma \vdash \text{get}(l, c) :_L Fx : A.Fe : (x =_A m).l \mapsto_A m}\textsc{Get}$$

$$\frac{\Gamma \vdash c :_L l \mapsto_A m \qquad \overline{\Gamma} \vdash n :_U A}{\Gamma \vdash \text{set}(l, c, n) :_L l \mapsto_A n}\textsc{Set}$$

Figure 11: State Programs

3.2.2 Laws for Free

4 Future Work

All of the theorems we have developed so far are purely syntactic in nature. But the natural correspondence between linear types and call-by-value semantics seem to hint at a deeper connection. Indeed, past works[28] have provided insight for the simply typed case. We would like to extend these results our dependent linear type theory.

5 Conclusion

References

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

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

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

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

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

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

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

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

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

[10] Coquand, T. Pattern matching with dependent types.

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

[12] 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.

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

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

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

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

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

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

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

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

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

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

[23] 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.

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

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

[26] Ore, C.-E. The extended calculus of constructions (ecc) with inductive types. *Information and Computation 99*, 2 (1992), 231–264.

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

[28] Pravato, A., Rocca, S. D., and Roversi, L. The call-by-value λ-calculus: a semantic investigation. *Mathematical Structures in Computer Science 9* (1999), 617–650.

[29] 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.

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

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

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

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

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

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