The Calculus of Linear Constructions

Qiancheng Fu

October 9, 2021

Abstract

The Calculus of Linear Constructions (CLC) is an extension of the Calculus of Constructions (CC) with linear types. Specifically, CLC extends CC with a hierarchy of linear universes that precisely controls the weakening and contraction of its term level inhabitants. We study the meta-theory of CLC, proving that it is a sound logical framework for reasoning about resource. CLC is backwards compatible with CC, allowing CLC to enjoy the 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 fine-grain propositions as types. Today, CC and its variations CIC [27] and ECC [26] lie at the core of popular proof assistants such as Coq [31], 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:

$$Money \to Camels + Marlboro$$

The customer will still maintain full ownership of their money after paying the vendor. 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 into CC [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 [33, 34] 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, 36, 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 [32] 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 $Money : \mathbb{N} \to L$, and a linear arrow $\multimap_L$ as follows.

$$Money\ 5 \multimap_L Camels + Marlboro$$

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 "minimally invasive" approach to extending CC with linear types, akin to Mazurak et al.'s work on System F [21]. This allows for a straightforward embedding of CLC into CC, and lifting of CC 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 CC. We construct a reduction preserving embedding of CLC into CC, 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 $Prop$ is the type of propositions in the style of CC.

$$
\begin{array}{lll}
s, t & ::= U \mid L & \text{sorts} \\
 & & \\
m, n, A, B, C & ::= U_i \mid L_i \mid Prop \mid x & \text{expressions} \\
 & \quad \mid \;\; (x : A) \to_s B \mid A \to_s B & \\
 & \quad \mid \;\; (x : A) \multimap_s B \mid A \multimap_s B & \\
 & \quad \mid \;\; \lambda x.n \mid m\ n &
\end{array}
$$

Figure 1: Syntax

A clear departure of our language from standard presentations of both linear type theory, and dependent type theory is the presence of four function types: $(x : A) \to_s B$, $A \to_s B$, $(x : A) \multimap_s B$, $A \multimap_s B$. The reason for these variants is that we have built the ! exponential of linear logic directly into the syntax of function types. 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 [35]. By integrating the exponential directly into function syntax, we give canonicity to ways that ! can be used. This allows us to derive substitution, and a direct embedding into CCω without needing extra machinery for manipulating exponential.

$$
\begin{aligned}
(\_ : A) \to_U B &\equiv\ !(!A \multimap !B) && (1)\\
(\_ : A) \to_L B &\equiv\ !(!A \multimap B) && (2)\\
A \to_U B &\equiv\ !(A \multimap !B) && (3)\\
A \to_L B &\equiv\ !(A \multimap B) && (4)\\
(\_ : A) \multimap_U B &\equiv\ !A \multimap !B && (5)\\
(\_ : A) \multimap_L B &\equiv\ !A \multimap B && (6)\\
A \multimap_U B &\equiv\ A \multimap !B && (7)\\
A \multimap_L B &\equiv\ A \multimap B && (8)
\end{aligned}
$$

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), (5), (6) if the co-domain B is dependent on arguments of the domain A.

2.2 Universes and Cumulativity

CLC features three sorts of universes, $Prop$, U, and L. $Prop$ is the impredicative universe propositions. U, and L 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 the universe which they belong. Intuitively, terms with types that occur within $Prop$, and U are unrestricted in their usage. Terms with types that occur within L 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. $Prop \preceq U_0 \preceq U_1 \preceq U_2 \preceq \cdots$
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
   (a) $(x : A_1) \to B_1 \preceq (x : A_2) \to B_2$
   (b) $(x : A_1) \multimap B_1 \preceq (x : A_2) \multimap B_2$
   (c) $A_1 \to B_1 \preceq A_2 \to B_2$
   (d) $A_1 \multimap B_1 \preceq A_2 \multimap B_2$

Figure 3 illustrates the structure of our universe hierarchy. Each linear universes L_i has U_{i+1} as its type, allowing functions to dependently 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 universe, and subsequently losing track of linearity.

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

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.

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

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

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

$$\frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma \qquad x \notin \Gamma_2}{\Gamma_1, x :_L A \ddagger \Gamma_2 \ddagger \Gamma, x :_L A}\textsc{Merge-L1} \qquad \frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Gamma \qquad x \notin \Gamma_1}{\Gamma_1 \ddagger \Gamma_2, x :_L A \ddagger \Gamma, x :_L A}\textsc{Merge-L2}$$

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 Type Formation

Typing judgments in our language take on the form $\Gamma \vdash m \overset{s}{:} A$ where s is an indexing sort that is either U or L. Intuitively, this judgment states that expression m has type A and sort s under context Γ. An expression of sort L is linear, and sort U is non-linear. Surprisingly, duplication of non-linear expression is not immediately allowed, only non-linear *values* can be safely duplicated. We will discuss this subtlety later in Section 2.7.2.

Type formation rules are presented in Figure 5. Rules worth discussing in detail are the L-Axiom and the rules for constructing various function types. For L-Axiom, the sort of all linear types L itself is of non-linear sort U. This means that a linear type can be freely used, avoiding the philosophical trappings discussed earlier.

The function types $(x : A) \to_s B$ and $A \to_s B$ represent non-linear functions, where the subscript s is the sort of co-domain B. Functions of these types contain no linear free variables, thus can be applied repeatedly without duplication of linear resources. The difference between the two is that $(x : A) \to_s B$ allows co-domain B to depended on input x of non-linear domain A. For $A \to_s B$, the domain A is linear so B is not allowed to depend on the function input.

The variants $(x : A) \multimap_s B$ and $A \multimap_s B$ represent linear functions. Functions of these types may contain linear free variables, thus cannot be applied repeated without duplication of linear variables. Similar to the non-linear versions, the difference between $(x : A) \multimap_s B$ and $A \multimap_s B$ lies in the allowance of dependency on function input for non-linear domain types.

$$\frac{\Gamma}{\Gamma \vdash U :_U U}\textsc{U-Axiom} \qquad \frac{\Gamma}{\Gamma \vdash L :_U U}\textsc{L-Axiom}$$

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma, x :_U A \vdash B :_U s}{\Gamma \vdash (x : A) \to_s B :_U U}\textsc{U-Prod} \qquad \frac{\Gamma \qquad \Gamma \vdash A :_U L \qquad \Gamma \vdash B :_U s}{\Gamma \vdash A \to_s B :_U U}\textsc{Arrow}$$

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma, x :_U A \vdash B :_U s}{\Gamma \vdash (x : A) \multimap_s B :_U L}\textsc{L-Prod} \qquad \frac{\Gamma \qquad \Gamma \vdash A :_U L \qquad \Gamma \vdash B :_U s}{\Gamma \vdash A \multimap_s B :_U L}\textsc{Lolli}$$

Figure 5: Type Formation

2.5 Term Formation

The term formations rules are presented in Figure 6.

The rules U-Var and L-Var state that a variable's type is determined by its context. In the case of L-Var, the linear variable x must be the only linear variable within its context. This enforcement of uniqueness eliminates the weakening rule for linear variables as redundant linear variables in the context will prevent the usage of L-Var.

The Conv rule is standard with regards to prior dependent type theory literature. Two types A and B are treated equivalently if they are $\beta\eta$-convertible. Our treatment of the convertibility relation $A \equiv B$ differs from standard due to our call-by-value semantics. We give a detailed account of this in the next section.

For the non-linear function formation rules U-λ_1 and U-λ_2, the context Γ is asserted to be pure. The purity of Γ ensures that no linear variables occur within function body n, allowing the function to be freely used without duplication of linear variables. For the linear function formation rules L-λ_1 and L-λ_2, the restriction on Γ's purity is lifted. Linear variables within Γ are allowed to occur freely within function body n. However, the tradeoff is that these linear functions may only be applied once, as multiple uses may result in duplication of its linear free variables.

In the application rules U-App-1 and L-App-1, the argument n is a non-linear expression that may contain linear free variables. If reduction is performed naively, substitution of n into the function body may cause duplication of these variables. Our operational semantics and value soundness lemma guarantee that substitution will not duplicate linear variables.

Furthermore, in the U-App-1 and L-App-1 rules, the input expression n is not directly substituted into the dependent co-domain B as this may introduce linear variables into types. Instead, a λ-abstraction is formed around B and applied to n. This delays β-reduction until n can be evaluated into a value.

$$\frac{\Gamma}{\Gamma, x :_U A \vdash x :_U A}\textsc{U-Var} \qquad \frac{\Gamma}{\Gamma, x :_L A \vdash x :_L A}\textsc{L-Var} \qquad \frac{\Gamma \vdash m \overset{s}{:} A \qquad A \equiv B}{\Gamma \vdash m \overset{s}{:} B}\textsc{Conv}$$

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

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

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

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

$$\frac{\Gamma_1 \vdash m :_U (x : A) \to_s B \qquad \Gamma_2 \vdash n :_U A \qquad \mathit{merge}\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n \overset{s}{:} (\lambda x.B)\ n}\textsc{U-App-1}$$

$$\frac{\Gamma_1 \vdash m :_U A \to_s B \qquad \Gamma_2 \vdash n :_L A \qquad \mathit{merge}\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n \overset{s}{:} B}\textsc{U-App-2}$$

$$\frac{\Gamma_1 \vdash m :_L (x : A) \multimap_s B \qquad \Gamma_2 \vdash n :_U A \qquad \mathit{merge}\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n \overset{s}{:} (\lambda x.B)\ n}\textsc{L-App-1}$$

$$\frac{\Gamma_1 \vdash m :_L A \multimap_s B \qquad \Gamma_2 \vdash n :_L A \qquad \mathit{merge}\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n \overset{s}{:} B}\textsc{L-App-2}$$

Figure 6: Term Formation

2.6 Reduction and Equality

Figure 7 presents the call-by-value operational semantics that we have eluded to. The rules defining the single step relation $\leadsto$ are completely standard.

For dependently typed languages, terms can appear at the type level. A definitional equality judgment is required to identify types beyond simple α-equivalence. This is usually accomplished by normalization and comparing normal forms. Due to the complications brought on by linearity and substitution hinted at in 2.5, standard normalization techniques cannot be directly applied. Unfortunately, the single step relation $\leadsto$ defined previously is not sufficient either as binders block evaluation. Following the footsteps of the Trellys project[30], we define a call-by-value parallel step relation $\leadsto_p$ that evaluates under binders. We use the transitive reflexive closure of $\leadsto_p$ to define definitional equality.

$$\frac{}{(\lambda x.n)\ v \rightsquigarrow [v/x]n}\text{S-U-}\beta \qquad \frac{}{(\lambda x.n)\ v \rightsquigarrow [v/x]n}\text{S-L-}\beta \qquad \frac{m \rightsquigarrow m'}{m\ n \rightsquigarrow m'\ n}\text{S-App-L}$$

$$\frac{n \rightsquigarrow n'}{v\ n \rightsquigarrow v\ n'}\text{S-App-R}$$

Figure 7: Single Step Reduction

$$\frac{m_1 \rightsquigarrow_p^* n \qquad m_2 \rightsquigarrow_p^* n}{m_1 \equiv m_2 : A}\text{Join} \qquad \frac{}{x \rightsquigarrow_p x}\text{P-Var} \qquad \frac{}{U \rightsquigarrow_p U}\text{P-U} \qquad \frac{}{L \rightsquigarrow_p L}\text{P-L}$$

$$\frac{n \rightsquigarrow_p n'}{\lambda x.n \rightsquigarrow_p \lambda x.n'}\text{P-}\lambda \qquad \frac{n \rightsquigarrow_p n'}{\lambda x.n \rightsquigarrow_p \lambda x.n'}\text{P-}\lambda \qquad \frac{m \rightsquigarrow_p m' \qquad n \rightsquigarrow_p n'}{m\ n \rightsquigarrow_p m'\ n'}\text{P-App}$$

$$\frac{n \rightsquigarrow_p n' \qquad v \rightsquigarrow_p v'}{(\lambda x.n)\ v \rightsquigarrow_p [v'/x]n'}\text{P-U-}\beta \qquad \frac{n \rightsquigarrow_p n' \qquad v \rightsquigarrow_p v'}{(\lambda x.n)\ v \rightsquigarrow_p [v'/x]n'}\text{P-L-}\beta$$

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

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

Figure 8: Equality and Parallel Reduction

2.7 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.

2.7.1 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 $m \rightsquigarrow_p m'$.*

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

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

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

2.7.2 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[35]. 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[35, 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* $\Gamma_1, x \overset{s}{:} A \vdash m \overset{t}{:} B$ *and* $\Gamma_2 \vdash v \overset{s}{:} A$*, if there exists* Γ *such that merge* $\Gamma_1\ \Gamma_2\ \Gamma$ *is defined, then* $\Gamma \vdash [v/x]m \overset{t}{:} [v/x]B$.

2.7.3 Type Soundness

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

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

Theorem 2.6. *Preservation. For* $\Gamma \vdash m \overset{s}{:} A$*, if* $m \rightsquigarrow_p n$ *then* $\Gamma \vdash n \overset{s}{:} A$.

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 9 with their Introduction and Elimination rules formalized in Figure 10 and Figure 11 respectively. Reduction for data types obey the same call-by-value semantics as outlined in 2.6. 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 9: Data Formation

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

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

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

Figure 10: Data Introduction

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

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

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

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

Figure 11: Data Elimination

3.2 Imperative Programming

With the addition of data types, axioms for imperative programming can be added to the language. In Figure 12 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

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma \vdash m :_U A \qquad \Gamma \vdash l :_U \mathbb{N}}{\Gamma \vdash l \mapsto_A m :_U L}\textsc{Capability}$$

$$\frac{\Gamma \qquad \Gamma \vdash A :_U U \qquad \Gamma \vdash m :_U A}{\Gamma \vdash \mathrm{new}(m) :_L Fl : \mathbb{N}.l \mapsto_A m}\textsc{New} \qquad \frac{\Gamma \qquad \Gamma \vdash c :_L l \mapsto_A m}{\Gamma \vdash \mathrm{free}(c) :_U \top}\textsc{Free}$$

$$\frac{\Gamma \vdash c :_L l \mapsto_A m}{\Gamma \vdash \mathrm{get}(l, c) :_L Fx : A.Fe : (x =_A m).l \mapsto_A m}\textsc{Get} \qquad \frac{\Gamma \vdash c :_L l \mapsto_A m \qquad \overline{\Gamma} \vdash n :_U A}{\Gamma \vdash \mathrm{set}(l, c, n) :_L l \mapsto_A n}\textsc{Set}$$

Figure 12: 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] Sjöberg, V., Casinghino, C., Ahn, K. Y., Collins, N., Eades, H., Fu, P., Kimmell, G., Sheard, T., Stump, A., and Weirich, S. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In *MSFP* (2012).

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

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

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

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

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

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