The Calculus of Linear Constructions

Qiancheng Fu

October 7, 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, with proofs formalized in the Coq Proof Assistant. We extend CLC with linear inductive types and show that it is straightforward to use CLC as a programming language for the safe manipulation of mutable data structures, with applications to verified functional programming in GC-less environments.

1 Introduction

The Calculus of Constructions (CC) is a dependent type theory introduced by Coquand and Huet in their landmark work [9]. In CC types can depend on terms, allowing one to write fine-grain propositions as types. Today, CC and its variations CIC [24] and ECC [23] lie at the core of popular proof assistants such as Coq [28], Agda [22], Lean [10], and others. These theorem provers have found great success in the fields of software verification [17, 7], and constructive mathematics [14, 5]. 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. On the other hand, resource must respect the conservation principle: resources are constant through change. Users of proof assistants based on CC often embed external logics into CC to provide additional reasoning principles to prove theorems concerning resources. The 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 [13]. 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 [29, 30] 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 [3] 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 [20, 32, 4] featuring linear types have been implemented in practice, allowing programmers to write memory safe software. The success of integrating Linear Logic with simple type theory exposes a tantalizing new frontier of integrating linearity with a richer type theory.

Cervesato and Pfenning extends the Edinburgh Logical Framework with linear types [15, 6], being the first to demonstrate that dependent types and linear types can coexist within a type theory. Vákár gives a categorical semantics for linear dependent types. Krishnaswami et al. present a dependent linear type theory [16] based on Benton's early work of mixed linear and non-linear

calculus, demonstrating the ability to internalize imperative programming the style of Hoare Type Theory [21]. 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 [19], Atkey [2] uses semi-ring annotations to track variable occurrence, simulating irrelevance, linear, and affine types within a unified framework.

2 Core Type Theory

In the efforts of integrating dependency and linearity, there are two predominant schools of thought: *Types can only depended on non-linear terms* and *Types can depend on all terms*.

Advocates of restricted dependency argue that the restriction is both intuitive and a necessity. A natural example of linear types depending on a non-linear terms is that of the length indexed random access array. Term dependency on array length statically prevents out-of-bounds array access whilst linearity ensures proper memory management. In the converse situation of types depending on linear terms, one immediately faces a dilemma: *Do linear terms at the type level possess weakening and contraction?*.

If one answers yes, then the linear terms at the type level are defacto non-linear. In this case, a language with restricted dependency can simulate linear dependency by depending on non-linear terms that reflect the shape of linear terms[12], erasing the advantage of increased type level expressivity. All prior works featuring linear dependency fall into this category, for good reason: *the alternative is worse*.

If type level linear terms do not possess weakening and contraction, types with linear dependencies themselves are linear. This breaks the very notion of the typing judgment as even α-equivalent terms are not allowed to possess the same type. Though the idea of such a type system is deeply interesting, the philosophical burden is prohibitive.

Our language falls firmly in the former category. Besides the reasons listed above, restricted dependency gives a clear distinction between linear and non-linear terms, a property we believe will aid in the compilation to high performance code.

2.1 Syntax

The syntax of the core type theory is presented in Figure 1. Our type theory contains two sorts, U and L for denoting the sort of non-linear types and linear types respectively. An important fact to note is that our language possess the "Type-in-Type" axiom in the form of $\Gamma \vdash U \overset{U}{:} U$ which is well known to be logically unsound. We chose this design deliberately as we are willing to sacrifice logical soundness for increased expressivity and convenience. We forsee no issue in extending the core language with a hierarchy of sorts to enforce logical soundness.

s, t	$::= U \mid L$	sorts
m, n, A, B, C	$::= U \mid L \mid x$ $\mid (x : A) \to_s B \mid A \to_s B$ $\mid (x : A) \multimap_s B \mid A \multimap_s B$ $\mid \lambda x.n \mid m\ n$	expressions
v	$::= U \mid L \mid x$ $\mid (x : A) \to_s B \mid A \to_s B$ $\mid (x : A) \multimap_s B \mid A \multimap_s B$ $\mid \lambda x.n$	values

Figure 1: Syntax

A clear departure of our language from standard presentations of 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 the decoupling of function linearity from the linearity of function domain and co-domain. Instead, the linearity of functions is determined by the linearity of the closure it forms. For example, a function with linear input and output type can itself be non-linear if its free variables have non-linear type, indicating this function can be applied repeatedly without duplication of linear variables. In practice, these function types can be consolidated to just $(x : A) \to_s B$ and $(x : A) \multimap_s B$ by treating $A \to_s B$ and $A \multimap_s B$ as non-binding special cases. For the sake of clarity, we leave them as four.

2.2 Context and Structural Judgments

The context of our language employs a mixed linear/non-linear representation in the style of Luo[18]. Variables in the context are annotated to indicate whether they are linear or non-linear. A non-linear variable is annotated as $\Gamma, x \overset{U}{:} A$, whereas a linear variable is annotated as $\Gamma, x \overset{L}{:} A$.

We define a *merge* relation to combine two mixed contexts Γ_1 and Γ_2 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. The definition of *merge* allows contraction for non-linear variables whilst forbidding it for linear ones.

An auxiliary judgment **pure** is defined to assert that a context Γ does not contain linear variables. In other words, all variables found in a pure context are of the form $\Gamma, x \overset{U}{:} A$. A restriction function $\overline{\Gamma}$ is defined that removes all linear variables from context Γ. The result of restriction is a pure subset of the original context.

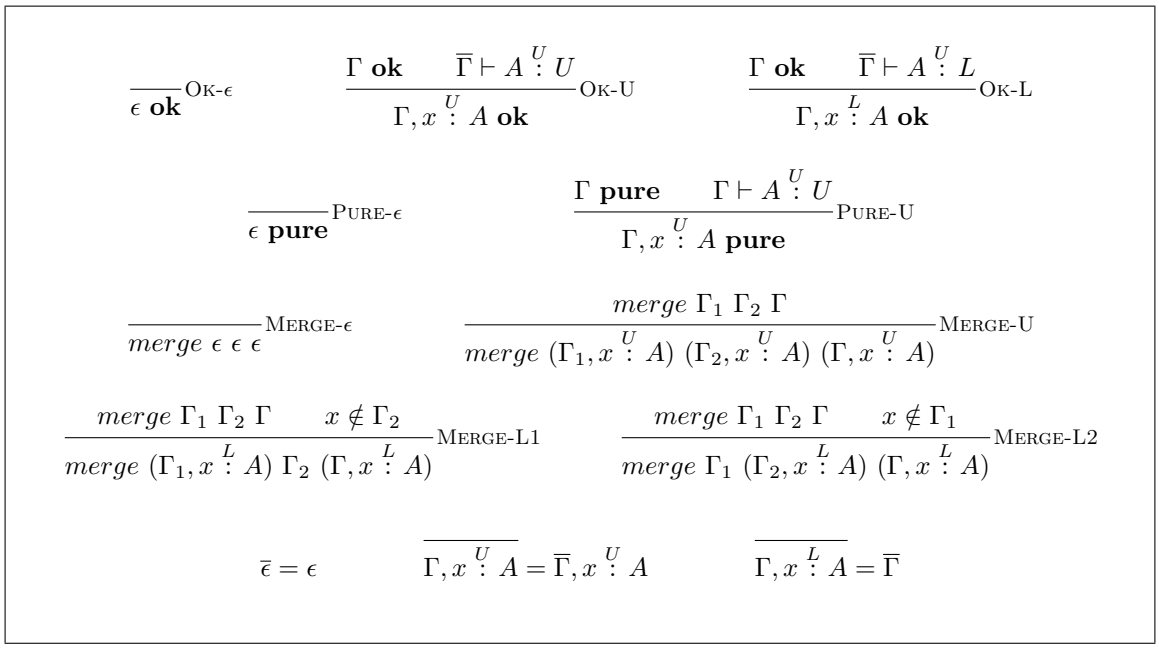

Figure 2: Structural Judgments

2.3 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 2.6.2.

Type formation rules are presented in Figure 3. 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.

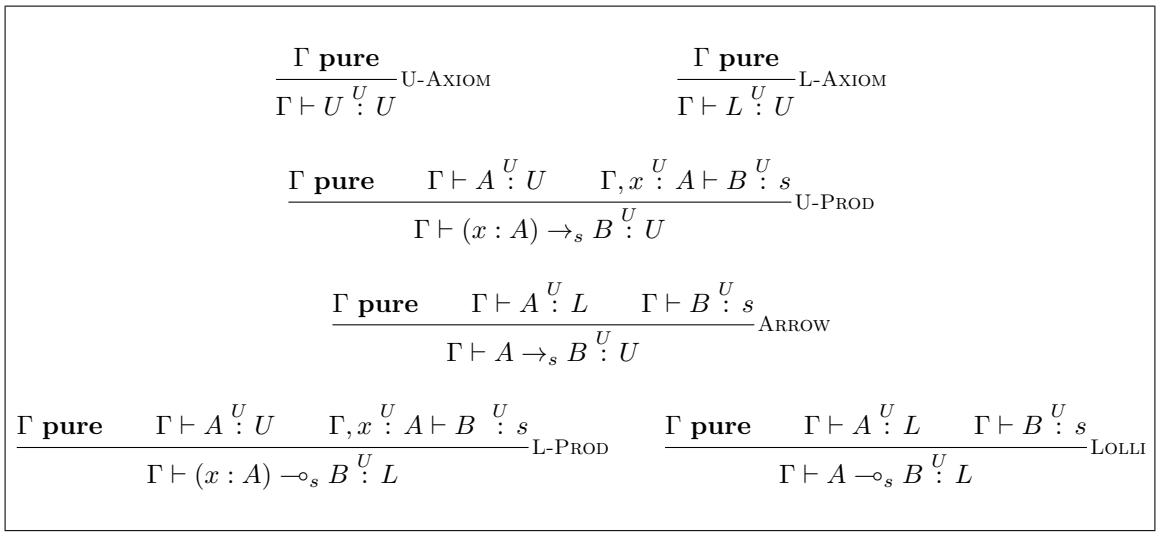

Figure 3: Type Formation

2.4 Term Formation

The term formations rules are presented in Figure 4.

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.

$$\frac{\Gamma \textbf{ pure}}{\Gamma, x \overset{U}{:} A \vdash x \overset{U}{:} A}\text{U-VAR} \qquad \frac{\Gamma \textbf{ pure}}{\Gamma, x \overset{L}{:} A \vdash x \overset{L}{:} A}\text{L-VAR} \qquad \frac{\Gamma \vdash m \overset{s}{:} A \qquad A \equiv B}{\Gamma \vdash m \overset{s}{:} B}\text{CONV}$$

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

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

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

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

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

$$\frac{\Gamma_1 \vdash m \overset{U}{:} A \to_s B \qquad \Gamma_2 \vdash n \overset{L}{:} A \qquad \mathit{merge}\ \Gamma_1\ \Gamma_2\ \Gamma}{\Gamma \vdash m\ n \overset{s}{:} B}\text{U-APP-2}$$

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

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

Figure 4: Term Formation

2.5 Reduction and Equality

Figure 5 presents the call-by-value operational semantics that we have eluded to. The rules defining the single step relation $\rightsquigarrow$ 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 2.4, standard normalization techniques cannot be directly applied. Unfortunately, the single step relation $\rightsquigarrow$ defined previously is not sufficient either as binders block evaluation. Following the footsteps of the Trellys project[27], we define a call-by-value parallel step relation $\rightsquigarrow_p$ that evaluates under binders. We use the transitive reflexive closure of $\rightsquigarrow_p$ 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 5: 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 6: Equality and Parallel Reduction

2.6 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[26] library.

2.6.1 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.*

2.6.2 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[31]. 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[31, 16]. Another is to ban non-linear expressions from containing linear variables[6, 18, 2].

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* $\Gamma \vdash v \overset{U}{:} A$ *then* Γ ***pure****.*

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* $\Gamma \vdash [v/x]m \overset{t}{:} [v/x]B$*.*

2.6.3 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[11] similar to Coq or Agda[22]. A pattern matching construct[8] 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 7 with their Introduction and Elimination rules formalized in Figure 8 and Figure 9 respectively. Reduction for data types obey the

same call-by-value semantics as outlined in 2.5. Other standard data types that are not covered here are $\mathbb{N}$ for natural numbers and $\top$ for the unit type.

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

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

Figure 7: Data Formation

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

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

$$\frac{\Gamma_1 \vdash m \overset{L}{:} A \qquad \Gamma_2 \vdash n \overset{L}{:} B \qquad merge\ \Gamma_1\Gamma_2\Gamma}{\Gamma \vdash \langle m,n\rangle \overset{L}{:} A \otimes B}\otimes\text{-INTRO} \qquad \frac{\Gamma \textbf{ pure} \qquad \Gamma \vdash n \overset{U}{:} A}{\Gamma \vdash \text{refl } n \overset{U}{:} n =_A n}$$

Figure 8: Data Introduction

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

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

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

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

Figure 9: Data Elimination

3.2 Imperative Programming

With the addition of data types, axioms for imperative programming can be added to the language. In Figure 10 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

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

$$\frac{\Gamma \textbf{ pure} \qquad \Gamma \vdash A \overset{U}{:} U \qquad \Gamma \vdash m \overset{U}{:} A}{\Gamma \vdash \text{new}(m) \overset{L}{:} Fl : \mathbb{N}.l \mapsto_A m}\textsc{New} \qquad \frac{\Gamma \textbf{ pure} \qquad \Gamma \vdash c \overset{L}{:} l \mapsto_A m}{\Gamma \vdash \text{free}(c) \overset{U}{:} \top}\textsc{Free}$$

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

Figure 10: 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[25] 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] 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] Benton, N. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *CSL* (1994).

[4] 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).

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

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

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

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

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

[10] 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.

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

[12] Fu, P., Kishida, K., and Selinger, P. Linear dependent type theory for quantum programming languages. *CoRR abs/2004.13472* (2020).

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

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

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

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

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

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

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

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] Schäfer, S., Tebbi, T., and Smolka, G. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015* (Aug 2015), X. Zhang and C. Urban, Eds., LNAI, Springer-Verlag.

[27] 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).

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

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

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

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

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