Dependent Linear Type Theory for Resource Aware Certified Programming

Qiancheng Fu

September 10, 2021

Abstract

TBD

1 Introduction

In the field of certified programming, it is of paramount importance for the specification language to precisely characterize the behavior of programs. For this reason, dependent type theory enjoys great success as the type level language of specifications is exactly the same as the term level programming language, giving tremendous power and control to specification writing. The verification achievements carried out by proof assistants such as Coq or F* is a testament how much dependent type theory has accomplished.

Dependent type theory is however, not without its flaws. Due to its origins as a logical foundation for mathematics, dependent type theory lacks facilities to reason about resource, a concept that is largely absent from mathematics but ever present in computer science. Objects in dependent type theory can be freely duplicated or deleted by appealing to the structural contraction and weakening rules, this goes against the conservation principle of resource. Users of dependent type theories are often required to laboriously embed memory models and program logics into type theory to effectively reason about resource. As language designers, we hope to alleviate this shortcoming of dependent type theory.

In the disparate world of substructural logic, Girard introduces Linear Logic in his seminal work. Linear Logic restricts weakening and contraction rules of classical logic, giving rise to an elegant formal foundation for reasoning about resource. Abramsky and Wadler notice the opportunities presented by Linear Logic for resource aware programming, pioneering the development of simple linear type theory. Linear type theory enforces linearity on variable usage, which conservatively subsumes resource usage. This reduces the problem of resource reasoning to linear type checking. Languages based on linear type systems have shown the capacity to safely manipulate imperative data structures. The successful extraction of type theory from Linear Logic offers a tantalizing hint to overcoming the challenges of resource reasoning in dependent type theory: integrate linear types.

The earliest work on integrating dependent type theory and linear type theory was carried out by Cervesato and Pfenning, extending the Edinburgh LF with linearity. Xi extends DML dependent types with linear types in the ATS programming language. Krishnaswami et al. develop LNL_D based on Benton's LNL calculus for simple linear types, allowing term dependency on its non-linear fragment. The authors of LNL_D demonstrate its capacity for certified imperative programming in the style of L3 and Hoare Type Theory. Luo et al. introduce the notion of essential linearity, developing the first type theory that allows types to depend on linear terms. Atkey's QTT, based on initial ideas of McBride, neatly reduces dependency and linearity checking to checking constraints defined on semi-rings. Idris 2 is a full blown programming language that implements QTT as its core type system.

The lack of weakening and contraction rules for linear types present unique difficulties when integrating full spectrum dependent types, often making dependent linear types challenging or unsatisfactory for practical use. $\mathrm{LNL_D}$ requires use of explicit modality maps F and G to alternate between its linear and non-linear fragment, computation requiring back and forth applications of F and G quickly explode in complexity. Terms in Luo's type theory are not intrinsically linear, linearity is solely determined by annotations. The unfortunate side-effect of this design choice is that linear computations never return non-linear values, contradicting the intuition of retrieving a non-linear int from a linear reference. QTT faces the same drawbacks as Luo due to linearity being determined by similar semi-ring annotations.

We present a full spectrum dependent linear type theory that provides a more direct approach to enforce linearity, eschewing the use of modality operators or annotations. We believe that linearity is an intrinsic property completely determined by the type of a term. To accomplish this, our type system uses two class of type universes in the style of Krishnaswami, L for linear and U for non-linear. Due to the abandonment of modalities, we use two distinct functions types $(x : A) \to B$ for non-linear functions and $(x : A) \multimap B$ for linear functions. We formalize a call-by-value operational semantics and prove the type soundness of our system. To further demonstrate the applications of dependent linear types, we extend our type system with imperative programming features, leveraging both dependency and linearity for certified imperative programming. We augment our soundness theorem to account for these extensions, showing that imperative resources are properly managed. We have implemented a prototype of our language extended with other features for practicality and ergonomics.

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, 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 may not possess the same type. Though the idea of such a type system is deeply interesting, the philosophical burden is prohibitive.

Our language firmly falls 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 universes, $\mathbf{U}$ and $\mathbf{L}$ for denoting the universe 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 $\Gamma; \cdot \vdash \mathbf{U} : \mathbf{U}$ 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 universes to enforce logical soundness.

$$
\begin{array}{lcll}
m, n, A, B, C & ::= & \mathbf{U} \mid \mathbf{L} \mid x & \text{expressions} \\
 & \mid & (x : A) \to B \mid A \to B & \\
 & \mid & (x : A) \multimap B \mid A \multimap B & \\
 & \mid & \lambda x.n \mid \hat{\lambda} x.n \mid m\ n & \\
 & & & \\
v & ::= & \mathbf{U} \mid \mathbf{L} \mid x & \text{values} \\
 & \mid & (x : A) \to B \mid A \to B & \\
 & \mid & (x : A) \multimap B \mid A \multimap B & \\
 & \mid & \lambda x.n \mid \hat{\lambda} x.n &
\end{array}
$$

Figure 1: Syntax

A clear departure of our language from standard presentations of type theory is the presence of four function types: $(x : A) \to B$, $A \to B$, $(x : A) \multimap B$, $A \multimap B$. 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 contraction of linear variables. In practice, these function types can be consolidated to just $(x : A) \to B$ and $(x : A) \multimap B$ by typechecking using Luo's mixed context and *merge* techniques. For the sake of clarity, we leave them as four.

2.2 Context and Structural Judgments

The context of our type theory is divided into two partitions, Γ for variables of non-linear type and Δ for variables of linear type. Weakening and contraction structural rules are permitted on Γ but forbidden on Δ. Structural rules for well-founded contexts are shown in Figure 2.

Types in our language are determined to be linear or non-linear by the universe they belong to. The typing judgment $\Gamma; \Delta \vdash A : \mathbf{U}$ asserts that A is a non-linear type under context $\Gamma; \Delta$, whereas the judgment $\Gamma; \Delta \vdash B : \mathbf{L}$ asserts that B is a linear type under context $\Gamma; \Delta$. In the rules for constructing well-founded contexts, care is taken to prevent Δ from leaking into typing judgments, restricting type dependency to Γ.

$$
\frac{}{\cdot\ ;\ \cdot \mathbf{ok}}\ \textsc{Empty-Ok}
\qquad
\frac{\Gamma; \cdot\ \mathbf{ok} \qquad \Gamma; \cdot \vdash A : \mathbf{U}}{\Gamma, x : A; \cdot\ \mathbf{ok}}\ \textsc{U-Ok}
\qquad
\frac{\Gamma; \Delta\ \mathbf{ok} \qquad \Gamma; \cdot \vdash A : \mathbf{L}}{\Gamma; \Delta, x : A\ \mathbf{ok}}\ \textsc{L-Ok}
$$

Figure 2: Structural Judgments

2.3 Type Formation

Type formation rules are presented in Figure 3. Rules worth detailed discussion are the L-Axiom and the rules for constructing various function types.

For L-Axiom, the universe of linear types itself belongs to the universe of non-linear types. This means that a linear type is itself a non-linear term, avoiding the philosophical trappings discussed above.

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

The variants $(x : A) \multimap B$ and $A \multimap B$ represent linear functions. Terms 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 $(x : A) \multimap B$ and $A \multimap B$ lies in the allowance of dependency on function input for non-linear domain types.

$$\frac{}{\Gamma; \cdot \vdash \mathbf{U} : \mathbf{U}}\textsc{U-Axiom} \qquad \frac{}{\Gamma; \cdot \vdash \mathbf{L} : \mathbf{U}}\textsc{L-Axiom} \qquad \frac{\Gamma; \cdot \vdash A : \mathbf{U}}{\Gamma; \cdot \vdash A\ \textbf{type}}\textsc{U-Type} \qquad \frac{\Gamma; \cdot \vdash A : \mathbf{L}}{\Gamma; \cdot \vdash A\ \textbf{type}}\textsc{L-Type}$$

$$\frac{\Gamma; \cdot \vdash A : \mathbf{U} \qquad \Gamma, x : A; \cdot \vdash B\ \textbf{type}}{\Gamma; \cdot \vdash (x : A) \to B : \mathbf{U}}\textsc{U-Prod} \qquad \frac{\Gamma; \cdot \vdash A : \mathbf{L} \qquad \Gamma; \cdot \vdash B\ \textbf{type}}{\Gamma; \cdot \vdash A \to B : \mathbf{U}}\textsc{Arrow}$$

$$\frac{\Gamma; \cdot \vdash A : \mathbf{U} \qquad \Gamma, x : A; \cdot \vdash B\ \textbf{type}}{\Gamma; \cdot \vdash (x : A) \multimap B : \mathbf{L}}\textsc{L-Prod} \qquad \frac{\Gamma; \cdot \vdash A : \mathbf{L} \qquad \Gamma; \cdot \vdash B\ \textbf{type}}{\Gamma; \cdot \vdash A \multimap B : \mathbf{L}}\textsc{Lolli}$$

Figure 3: Type Formation

2.4 Term Formation

Term formation rules are presented in Figure 4. As outlined previously, terms constructed by λ for non-linear function types $(x : A) \to B$ and $A \to B$ are only well type if their function bodies contain only non-linear variables listed in Γ. Conversely, terms constructed by $\hat{\lambda}$ for linear types $(x : A) \multimap B$ and $A \multimap B$ are allowed to contain both linear and non-linear variables listed in $\Gamma; \Delta$.

In classical dependent type theories such as Martin Löf Type Theory or Calculus of Constructions, strong normalization and confluence ensure that term normalization is agnostic to reduction strategy. Furthermore, the preservation theorem shows that reduction preserves the typing judgment of well-typed terms. The management of substructural contexts for dependent linear types complicates the direct use of these theorems. In the application rules U-App-1 and L-App-1, the argument n is a non-linear term that may contain free variables in linear context Δ. If a call-by-name reduction strategy is used, substitution of n into the function body may cause duplication of its linear free variables. Some existing literature ban non-linear arguments from containing linear free variables. We find this approach to be overly restrictive as it excludes the pattern of freely using retrieved data (non-linear) from references (linear) ubiquitous in imperative programs. Instead, we adjust the operational semantics to favor call-by-value reduction strategy. Our value soundness theorem guarantees non-linear values to contain only non-linear free variables. Thus, substitution will no longer duplicate the linear free variables found in arguments.

$$\frac{}{\Gamma, x : A; \cdot \vdash x : A}\textsc{U-Var} \qquad \frac{}{\Gamma; x : A \vdash x : A}\textsc{L-Var}$$

$$\frac{\Gamma; \Delta \vdash m : A \qquad \Gamma; \cdot \vdash A \equiv B : \mathbf{U}}{\Gamma; \Delta \vdash m : B}\textsc{U-Conv} \qquad \frac{\Gamma; \Delta \vdash m : A \qquad \Gamma; \cdot \vdash A \equiv B : \mathbf{L}}{\Gamma; \Delta \vdash m : B}\textsc{L-Conv}$$

$$\frac{\Gamma; \cdot \vdash (x : A) \to B : \mathbf{U} \qquad \Gamma, x : A; \cdot \vdash n : B}{\Gamma; \cdot \vdash \lambda x.n : (x : A) \to B}\text{U-}\lambda_1$$

$$\frac{\Gamma; \cdot \vdash A \to B : \mathbf{U} \qquad \Gamma; x : A \vdash n : B}{\Gamma; \cdot \vdash \lambda x.n : A \to B}\text{U-}\lambda_2$$

$$\frac{\Gamma; \cdot \vdash (x : A) \multimap B : \mathbf{L} \qquad \Gamma, x : A; \Delta \vdash n : B}{\Gamma; \Delta \vdash \hat{\lambda} x.n : (x : A) \multimap B}\text{L-}\lambda_1$$

$$\frac{\Gamma; \cdot \vdash A \multimap B : \mathbf{L} \qquad \Gamma; \Delta, x : A \vdash n : B}{\Gamma; \Delta \vdash \hat{\lambda} x.n : A \multimap B}\text{L-}\lambda_2$$

$$\frac{\Gamma; \Delta_1 \vdash m : (x : A) \to B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1, \Delta_2 : m\ n : (\lambda x.B)\ n}\textsc{U-App-1}$$

$$\frac{\Gamma; \Delta_1 \vdash m : A \to B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1, \Delta_2 : m\ n : B}\textsc{U-App-2}$$

$$\frac{\Gamma; \Delta_1 \vdash m : (x : A) \multimap B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1, \Delta_2 : m\ n : (\lambda x.B)\ n}\textsc{L-App-1}$$

$$\frac{\Gamma; \Delta_1 \vdash m : A \multimap B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1, \Delta_2 : m\ n : B}\textsc{L-App-2}$$

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 full spectrum 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 outlined in 2.3, standard normalization techniques cannot be directly applied. Unfortunately, the single step relation $\rightsquigarrow$ defined previously is not sufficient either as binders block full normalization. Following in the footsteps of the Trellys project, we define a call-by-value parallel step relation $\rightsquigarrow_p$. The $\rightsquigarrow_p$ relation is essentially the same as $\rightsquigarrow$ except that it normalizes under binders. We use the transitive reflexive closure of $\rightsquigarrow_p$ to define definitional equality.

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

Figure 5: Single Step Reduction

$$\frac{\Gamma;\Delta \vdash m_1 : A \qquad \Gamma;\Delta \vdash m_2 : A \qquad m_1 \rightsquigarrow_p^* n \qquad m_2 \rightsquigarrow_p^* n}{\Gamma;\Delta \vdash m_1 \equiv m_2 : A}$$

Figure 6: Equality

$$\frac{}{x \rightsquigarrow_p x}\,\textsc{P-Var} \qquad \frac{}{\mathbf{U} \rightsquigarrow_p \mathbf{U}}\,\text{P-U} \qquad \frac{}{\mathbf{L} \rightsquigarrow_p \mathbf{L}}\,\text{P-L} \qquad \frac{n \rightsquigarrow_p n'}{\lambda x.n \rightsquigarrow_p \lambda x.n'}\,\text{P-}\lambda \qquad \frac{n \rightsquigarrow_p n'}{\hat{\lambda} x.n \rightsquigarrow_p \hat{\lambda} x.n'}\,\text{P-}\hat{\lambda}$$

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

$$\frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{(x : A) \to B \rightsquigarrow_p (x : A') \to B'}\,\textsc{P-U-Prod} \qquad \frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{A \to B \rightsquigarrow_p A' \to B'}\,\textsc{P-Arrow}$$

$$\frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{(x : A) \multimap B \rightsquigarrow_p (x : A') \multimap B'}\,\textsc{P-L-Prod} \qquad \frac{A \rightsquigarrow_p A' \qquad B \rightsquigarrow_p B'}{A \multimap B \rightsquigarrow_p A' \multimap B'}\,\textsc{P-L-Lolli}$$

Figure 7: Parallel Reduction

2.6 Meta Theory

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 $m_2 \rightsquigarrow_p^* m'$.*

Lemma 2.3. *Value typing. If $\Gamma;\Delta \vdash v : A$ then $\Gamma;\cdot \vdash A : \boldsymbol{U}$ or $\Gamma;\cdot \vdash A : \boldsymbol{L}$.*

Lemma 2.4. *Value soundness. If $\Gamma;\Delta \vdash v : A$ and $\Gamma;\cdot \vdash A : \boldsymbol{U}$ then $\Delta = \cdot$.*