Dependent Linear Type Theory for Resource Aware Certified Programming

Qiancheng Fu

September 7, 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. 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 $\mathrm{LNL}_{\mathrm{D}}$ based on Benton's LNL calculus for simple linear types, allowing term dependency on its non-linear fragment. The authors of $\mathrm{LNL}_{\mathrm{D}}$ 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. 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 in the style of Pfenning. We formalize an 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 internalized 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

$$
\begin{array}{lcl}
x, y, z, f, g, h & \in & \text{variables} \\
m, n, A, B, C & \in & \text{expressions} \\
v & \in & \text{values} \\
\\
\text{expressions} & ::= & \mathbf{U} \mid \mathbf{L} \mid x \\
 & \mid & (x : A) \to B \mid A \to B \\
 & \mid & (x : A) \multimap B \mid A \multimap B \\
 & \mid & \lambda x.n \mid m\ n \\
\\
\text{values} & ::= & \mathbf{U} \mid \mathbf{L} \mid x \\
 & \mid & (x : A) \to B \mid A \to B \\
 & \mid & (x : A) \multimap B \mid A \multimap B \\
 & \mid & \lambda x.n
\end{array}
$$

Figure 1: Syntax

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

Figure 2: Structural Judgments

$$\frac{}{\Gamma;\cdot \vdash \mathbf{U} : \mathbf{U}} \qquad \frac{}{\Gamma;\cdot \vdash \mathbf{L} : \mathbf{U}} \qquad \frac{\Gamma;\cdot \vdash A : \mathbf{U}}{\Gamma;\cdot \vdash A\ \mathbf{type}} \qquad \frac{\Gamma;\cdot \vdash A : \mathbf{L}}{\Gamma;\cdot \vdash A\ \mathbf{type}}$$

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

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

Figure 3: Type Formation

$$\frac{}{\Gamma, x : A;\cdot \vdash x : A} \qquad \frac{}{\Gamma; x : A \vdash x : A} \qquad \frac{\Gamma;\Delta \vdash m : A \qquad \Gamma;\Delta \vdash A \equiv B}{\Gamma;\Delta \vdash m : B}$$

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

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

$$\frac{\Gamma;\Delta_1 \vdash m : (x : A) \to B \qquad \Gamma;\Delta_2 \vdash n : A}{\Gamma;\Delta_1,\Delta_2 : m\ n : [n/x]B} \qquad \frac{\Gamma;\Delta_1 \vdash m : A \to B \qquad \Gamma;\Delta_2 \vdash n : A}{\Gamma;\Delta_1,\Delta_2 : m\ n : B}$$

$$\frac{\Gamma;\Delta_1 \vdash m : (x : A) \multimap B \qquad \Gamma;\Delta_2 \vdash n : A}{\Gamma;\Delta_1,\Delta_2 : m\ n : [n/x]B} \qquad \frac{\Gamma;\Delta_1 \vdash m : A \multimap B \qquad \Gamma;\Delta_2 \vdash n : A}{\Gamma;\Delta_1,\Delta_2 : m\ n : B}$$

Figure 4: Term Formation

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

Figure 5: Equality

$$\frac{}{n \rightsquigarrow_p n} \qquad \frac{n \rightsquigarrow_p n'}{\lambda x.n \rightsquigarrow_p \lambda x.n'} \qquad \frac{m \rightsquigarrow_p m' \qquad n \rightsquigarrow_p n'}{m\ n \rightsquigarrow_p m'\ n'} \qquad \frac{n \rightsquigarrow_p n' \qquad v \rightsquigarrow_p v'}{(\lambda x.n)\ v \rightsquigarrow_p [v'/x]n'}$$

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

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

Figure 6: Parallel Reduction