Integrating Dependent and Linear Types

Neelakantan R. Krishnaswami
University of Birmingham
<n.krishnaswami@cs.bham.ac.uk>

Pierre Pradic
ENS Lyon
<pierre.pradic@ens-lyon.fr>

Nick Benton
Microsoft Research
<nick@microsoft.com>

Abstract

In this paper, we show how to integrate linear types with type dependency, by extending the linear/non-linear calculus of Benton to support type dependency.

Next, we give an application of this calculus by giving a proof-theoretic account of imperative programming, which requires extending the calculus with computationally irrelevant quantification, proof irrelevance, and a monad of computations. We show the soundness of our theory by giving a realizability model in the style of Nuprl, which permits us to validate not only the β-laws for each type, but also the η-laws.

These extensions permit us to decompose Hoare triples into a collection of simpler type-theoretic connectives, yielding a rich equational theory for dependently-typed higher-order imperative programs. Furthermore, both the type theory and its model are relatively simple, even when all of the extensions are considered.

Keywords Linear types, dependent types, intersection types, proof irrelevance, separation logic, Hoare triples

1. Introduction

Two of the most influential research directions in language design are substructural logic and dependent type theory.

Substructural logics like linear logic [18] and separation logic [38] offer fine-grained control over the structural rules of logic (i.e., contraction and weakening), which means types control not only how a value of a given type is used, but also how often it is used. This permits giving a logical account of the notion of *resource*, and has been tremendously useful in giving modular and accurate specifications to stateful and imperative programs.

On the other hand, dependent type theory extends simple type theory by permitting terms to occur in types, permitting the expression of concepts like equality in types. This enables giving types which give a very fine-grained specification of the functional behaviour of their inhabitants.

We would like to fully integrate dependent and linear type theory. Linear type theory would permit us to extend the Curry-Howard correspondence to (for example) *imperative* programs, and type dependency permits giving very precise types to describe the precise functional behavior of those programs. The primary difficulty we face is precisely the interaction between type dependency and the structural rules. Since linear types ensure a variable occurs only once in a program, and dependent types permit variables to occur in both types and terms, how should we count occurrences of variables in types?

The key observation underpinning our approach is that *we do not have to answer this question!* We build on the work of Benton [7], which formulated models of intuitionistic linear logic in terms of a monoidal adjunction $F \dashv G$ between a monoidal closed category $\mathbb{L}$ and a cartesian closed category $\mathbb{C}$. The exponential $!A$ then decomposes as the composition of the $G(F(A))$. Syntactically, such a model then corresponds to a pair of lambda calculi, one intuitionistic, and one linear, which are related by a pair of modal operators F and G.

In the linear/non-linear (LNL) calculus, we no longer have to view the intuitionistic function space $P \to Q$ through the lens of the Girard encoding $!P \multimap Q$. Instead, the intuitionistic arrow, $\to$, is a connective in its own right, independent of the linear function space, $\multimap$. Given such a separation, we can make the intuitionistic part of the language dependent, changing the type $X \to Y$ to the pi-type $\Pi x : X.\, Y$, without altering the linear function space $A \multimap B$. We then end up with a type theory in which linear types may depend on intuitionistic terms in quite sophisticated ways, but we never have to form a type dependent on an open linear term.

As an illustration, recall Girard's [19] classic motivating example of the resource reading of linear logic in which the proposition A represents spending a dollar, B represents obtaining a pack of Camels, and C obtaining a pack of Marlboro. The formula $A \multimap B\&C$ then expresses that one may spend a dollar and receive the option of a pack of Camels and of a pack of Marlboro, but only of these options may be exercised. In LNL_D, from that assumption one can type, for example, a machine M that takes n dollars and produces its choice of something you can buy:

$$M : \Pi n : \mathbb{N}.\ \mathsf{tuple}\ n\ A \multimap F m : \mathbb{N}, m' : \mathbb{N}, p : m + m' = n.\ \mathsf{tuple}\ m\ B \otimes \mathsf{tuple}\ m'\ C$$

where $\mathsf{tuple}\ n\ A$ is the (definable) n-fold linear tensor $A \otimes \cdots \otimes A$. The return type of the linear function space in the type of M is a linear dependent pair, which generalizes LNL's F modality for mapping intuitionistic values into linear ones. So M yields two integers m and m', a proof that their sum is n and an m-tuple of Bs together with an m'-tuple of Cs.

Contributions. In this paper, we make the following contributions:

- First, we describe a core type theory LNL_D that smoothly integrates both linearity and full type dependency, by ex-

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676969

$$\boxed{\Gamma \text{ ok}} \qquad \frac{}{\cdot \text{ ok}} \qquad \frac{\Gamma \text{ ok} \quad \Gamma \vdash X \text{ type}}{\Gamma, x : X \text{ ok}}$$

$$\boxed{\Gamma \vdash \Delta \text{ ok}} \qquad \frac{}{\Gamma \vdash \cdot \text{ ok}} \qquad \frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash A \text{ linear}}{\Gamma \vdash \Delta, a : A \text{ ok}}$$

$$\boxed{\Gamma \vdash X \text{ type}} \qquad \frac{\Gamma \vdash X : U_i}{\Gamma \vdash X \text{ type}} \qquad \boxed{\Gamma \vdash A \text{ linear}} \qquad \frac{\Gamma \vdash A : L_i}{\Gamma \vdash A \text{ linear}}$$

$$\boxed{\Gamma \vdash X \equiv Y \text{ type}} \qquad \frac{\Gamma \vdash X \equiv Y : U_i}{\Gamma \vdash X \equiv Y \text{ type}} \qquad \boxed{\Gamma \vdash A \equiv B \text{ linear}} \qquad \frac{\Gamma \vdash A \equiv B : L_i}{\Gamma \vdash A \equiv B \text{ linear}}$$

Figure 1. Structural judgements

$$\frac{}{\Gamma \vdash U_i : U_{i+1}} \qquad \frac{}{\Gamma \vdash L_i : U_{i+1}}$$

$$\frac{\Gamma \vdash X : U_i \quad \Gamma, x : X \vdash Y : U_i}{\Gamma \vdash \Pi x : X.\ Y : U_i} \qquad \frac{\Gamma \vdash X : U_i \quad \Gamma, x : X \vdash Y : U_i}{\Gamma \vdash \Sigma x : X.\ Y : U_i}$$

$$\frac{}{\Gamma \vdash 1 : U_i} \qquad \frac{\Gamma \vdash A : L_i}{\Gamma \vdash G\,A : U_i}$$

$$\frac{\Gamma \vdash X : U_i \quad \Gamma \vdash e : X \quad \Gamma \vdash e' : X}{\Gamma \vdash e =_X e' : U_i}$$

$$\frac{}{\Gamma \vdash I : L_i} \qquad \frac{\Gamma \vdash A : L_i \quad \Gamma \vdash B : L_i}{\Gamma \vdash A \otimes B : L_i} \qquad \frac{\Gamma \vdash A : L_i \quad \Gamma \vdash B : L_i}{\Gamma \vdash A \multimap B : L_i}$$

$$\frac{\Gamma \vdash X : U_i \quad \Gamma, x : X \vdash A : L_i}{\Gamma \vdash \Pi x : X.\ A : L_i} \qquad \frac{\Gamma \vdash X : U_i \quad \Gamma, x : X \vdash A : L_i}{\Gamma \vdash Fx : X.\ A : L_i}$$

$$\frac{}{\Gamma \vdash \top : L_i} \qquad \frac{\Gamma \vdash A : L_i \quad \Gamma \vdash B : L_i}{\Gamma \vdash A \,\&\, B : L_i}$$

Figure 2. Type Well-formedness

$$\frac{}{\Gamma, x : X, \Gamma' \vdash x : X} \qquad \frac{\Gamma \vdash e : Y \quad \Gamma \vdash X \equiv Y \text{ type}}{\Gamma \vdash e : X} \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash e : X \quad \Gamma \vdash e' : [e/x]Y}{\Gamma \vdash (e, e') : \Sigma x : X.\ Y} \qquad \frac{\Gamma \vdash e : \Sigma x : X.\ Y}{\Gamma \vdash \pi_1\, e : X}$$

$$\frac{\Gamma \vdash e : \Sigma x : X.\ Y}{\Gamma \vdash \pi_2\, e : [\pi_1\, e/x]Y}$$

$$\frac{\Gamma \vdash \Pi x : X.\ Y \text{ type} \quad \Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x.\ e : \Pi x : X.\ Y}$$

$$\frac{\Gamma \vdash e : \Pi x : X.\ Y \quad \Gamma \vdash e' : X}{\Gamma \vdash e\ e' : [e'/x]Y}$$

$$\frac{\Gamma \vdash e \equiv e' : X}{\Gamma \vdash \text{refl} : e =_X e'} \qquad \frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash G\,e : G\,A}$$

Figure 3. Intuitionistic Typing

tending the linear/non-linear calculus [7, 8]. Our type theory is an extensional type theory, which means that we consider the full βη-theory of the language, and furthermore, it also has a universe hierarchy, meaning that we support "full-spectrum" dependent types including large eliminations (i.e., computing types from terms).

Our approach to linear dependent type theory gives a simple proof-theoretic explanation of how the ad-hoc restrictions on the interaction of linearity in earlier approaches (such as Linear LF [10]) arise, by showing how they can be modelled in LNL_D.

- Second, we show how to turn this calculus into a language for dependently-typed imperative programming (in the style of separation logic). The general idea is to combine linearity and type dependency to turn separation-logic style specifications into types, and to take imperative programs to be inhabitants of those types.

  Ensuring that programs have the appropriate operational behavior requires a number of extensions to the type system, including pointer types, implicit quantification (in the style of intersection and union types), and proof irrelevance for linear types.
- Once this is done, we not only have a very expressive type system for higher-order imperative programs, but we also can systematically give a natural equational theory for them via the βη-theory for each of the type constructors we use.
- Next, we give a realizability model in the style of Nuprl [20] to show the consistency of our calculus. Our model is very simple and easy to work with, but nevertheless simply and gracefully justifies the wide variety of the extensions we consider.
- Finally, we give a small implementation of our extended type theory, illustrating that it is possible to implement all of these constructions.

2. Core Dependent Linear Type Theory

In this section, we describe the core dependent linear/nonlinear calculus. The full grammar of terms and types is given later, in Figure 10, but here we present the calculus rule by rule.

The basic structural judgements are given in Figure 1. The judgement Γ ok is the judgement that a context Γ of *intuitionistic hypotheses* is well-formed. As is standard, a well-formed context is either empty, or a context $\Gamma', x : X$ where Γ' is well-formed and X is an intuitionistic type well-formed in Γ. The judgement $\Gamma \vdash \Delta$ ok says that Δ is a well-formed *linear context*, whose linear types may depend (only) upon the intuitionistic variables in Γ. The judgements $\Gamma \vdash X$ type and $\Gamma \vdash A$ linear judge when a term is an intuitionistic or linear type, respectively. The equality judgements $\Gamma \vdash X \equiv Y$ type and $\Gamma \vdash A \equiv B$ linear determine whether two intuitionistic or linear types are definitionally equal, respectively.

Both $\Gamma \vdash X$ type and $\Gamma \vdash A$ linear work by immediately appealing to an ordinary typing judgement for an intuitionistic universe U_i and a linear universe L_i. The type well-formedness rules are given in Figure 2, and the introduction and elimination rules are given in Figure 3, with the judgement $\Gamma \vdash e : X$.

Both linear and intuitionistic universes L_i and U_i are elements of the larger universe U_{i+1}. Note that L_i is an element of U_{i+1} — this is because a linear *type* is an intuitionistic *term*, since we do not wish to restrict how often type variables ranging over linear types can occur. (That is, we want polymorphic types like $\Pi\alpha : L_i.\ \alpha \multimap \alpha$ to be well-formed.)

$$\frac{}{\Gamma; a : A \vdash a : A} \qquad \frac{\Gamma; \Delta \vdash e : B \qquad \Gamma \vdash A \equiv B \text{ linear}}{\Gamma; \Delta \vdash e : A}$$

$$\frac{}{\Gamma; \cdot \vdash () : I} \qquad \frac{\Gamma; \Delta \vdash e : I \qquad \Gamma; \Delta' \vdash e' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ () = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Gamma; \Delta \vdash e : A \qquad \Gamma; \Delta' \vdash e' : B}{\Gamma; \Delta, \Delta' \vdash (e, e') : A \otimes B}$$

$$\frac{\Gamma; \Delta \vdash e : A \otimes B \qquad \Gamma; \Delta', a : A, b : B \vdash e' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ (a, b) = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Gamma; \Delta, a : A \vdash e : B}{\Gamma; \Delta \vdash \lambda a.\ e : A \multimap B} \qquad \frac{\Gamma; \Delta \vdash e : A \multimap B \qquad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash e\ e' : B}$$

$$\frac{\Gamma, x : X; \Delta \vdash e : A}{\Gamma; \Delta \vdash \hat{\lambda} x.\ e : \Pi x : X.\ A} \qquad \frac{\Gamma; \Delta \vdash e : \Pi x : X.\ A \qquad \Gamma \vdash e' : X}{\Gamma; \Delta \vdash e\ e' : [e'/x]A}$$

$$\frac{}{\Gamma; \Delta \vdash () : \top}$$

$$\frac{\Gamma; \Delta \vdash e_1 : A_1 \qquad \Gamma; \Delta \vdash e_2 : A_2}{\Gamma; \Delta \vdash (e_1, e_2) : A_1 \,\&\, A_2} \qquad \frac{\Gamma; \Delta \vdash e : A \,\&\, B}{\Gamma; \Delta \vdash \pi_1\, e : A}$$

$$\frac{\Gamma; \Delta \vdash e : A \,\&\, B}{\Gamma; \Delta \vdash \pi_2\, e : B}$$

$$\frac{\Gamma \vdash e : X \qquad \Gamma; \Delta \vdash t : [e/x]A}{\Gamma; \Delta \vdash \mathsf{F}\,(e, t) : \mathsf{F}x : X.\ A}$$

$$\frac{\Gamma; \Delta \vdash e : \mathsf{F}x : X.\ A \qquad \Gamma, x : X; \Delta', a : A \vdash e' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ \mathsf{F}\,(x, a) = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Gamma \vdash e : \mathsf{G}\,A}{\Gamma; \cdot \vdash \mathsf{G}^{-1}\, e : A}$$

Figure 4. Linear Typing

The intuitionistic types we consider are the dependent function space $\Pi x : X.\ Y$, with a lambda-abstraction and function application as its introduction and elimination rules; the dependent pair $\Sigma x : X.\ Y$, with pairing (e, e') as its introduction, and the projective (or "strong") eliminations $\pi_1\, e$ and $\pi_2\, e$; the unit type 1; and the equality type $e =_X e'$. Equality is introduced with refl, but has no explicit elimination form, since we are giving an extensional type theory with equality reflection. These rules are all standard for dependent type theory. Similarly, it is possible to add inductive types (such as natural numbers) to the calculus, including support for large eliminations, though we omit them from the rules in the paper for space reasons (rules and proofs are in the companion tech report).

The first place that linearity arises is with the the adjoint embedding of linear types into intuitionistic types $\mathsf{G}\,A$. Its (intuitionistic) introduction rule is $\mathsf{G}\,e$, which checks that e has linear type A, in the empty linear context. (Intuitively, this reflects the fact that closed linear terms can be freely duplicated.) It is worth noting that our intuitionistic types are also essentially the same types as are found in the non-dependent LNL calculus, only we have changed $X \to Y$ to a pi-type, and $X \times Y$ to a sigma-type.

The typing judgement $\Gamma; \Delta \vdash e : A$ for linear terms says that e has type A, assuming intuitionistic variables Γ and linear variables Δ. The well-formedness and typing rules for the MALL connectives $(I, \otimes, \multimap, \&, \top)$ are all standard. The elimination form of the type $\mathsf{G}\,A$ is $\mathsf{G}^{-1}\, e$, which yields an A in an empty linear context, which is the same as in Benton [7].

$$\boxed{\Gamma \vdash e \equiv e' : X} \qquad \boxed{\Gamma; \Delta \vdash t \equiv t' : A}$$

$$\frac{\Gamma \vdash p : e =_X e'}{\Gamma \vdash e \equiv e' : X}$$

$$\frac{}{\Gamma \vdash (\lambda x.\ e)\ e' \equiv [e'/x]e : X} \qquad \frac{}{\Gamma \vdash e \equiv \lambda x.\ e\ x : \Pi x : X.\ Y}$$

$$\frac{}{\Gamma \vdash \pi_1\, (e, e') \equiv e : X} \qquad \frac{}{\Gamma \vdash \pi_2\, (e, e') \equiv e' : X}$$

$$\frac{}{\Gamma \vdash e \equiv (\pi_1\, e, \pi_2\, e) : \Sigma x : X.\ Y}$$

$$\frac{}{\Gamma \vdash e \equiv e' : 1}$$

$$\frac{}{\Gamma \vdash \mathsf{G}\,(\mathsf{G}^{-1}\, e) \equiv e : \mathsf{G}\,A}$$

$$\frac{}{\Gamma; \cdot \vdash \mathsf{G}^{-1}\,(\mathsf{G}\, t) \equiv t : A} \quad \frac{}{\Gamma; \Delta \vdash (\lambda x.\ e)\ e' \equiv [e'/x]e : C}$$

$$\frac{}{\Gamma; \Delta \vdash e \equiv \lambda x.\ e\ x : A \multimap B}$$

$$\frac{}{\Gamma; \Delta \vdash (\hat{\lambda} x.\ e)\ e' \equiv [e'/x]e : C} \qquad \frac{}{\Gamma; \Delta \vdash e \equiv \hat{\lambda} x.\ e\ x : \Pi x : X.\ A}$$

$$\frac{}{\Gamma; \Delta \vdash e \equiv e' : \top}$$

$$\frac{}{\Gamma; \Delta \vdash \pi_1\, (e, e') \equiv e : A} \qquad \frac{}{\Gamma; \Delta \vdash \pi_1\, (e, e') \equiv e' : B}$$

$$\frac{}{\Gamma; \Delta \vdash e \equiv (\pi_1\, e, \pi_2\, e) : A \,\&\, B}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ () = ()\ \mathsf{in}\ e \equiv e : C}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ () = t\ \mathsf{in}\ [()/x]t' \equiv [t/x]t' : C}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ (a, b) = (t_1, t_2)\ \mathsf{in}\ t' \equiv [t_1/a, t_2/b]t' : C}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ (a, b) = t\ \mathsf{in}\ [(a, b)/x]t' \equiv [t/x]t' : C}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ \mathsf{F}\,(x, a) = \mathsf{F}\,(e, t)\ \mathsf{in}\ t' \equiv [e/x, t/a]t' : C}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let}\ \mathsf{F}\,(x, a) = t\ \mathsf{in}\ [\mathsf{F}\,(x, a)/y]t' \equiv [t/y]t' : C}$$

Figure 5. βη-Equality

We depart from propositional LNL when we reach the dependent version of the other adjoint connective. Instead of a unary modal connective $\mathsf{F}(X)$, we introduce the *linear dependent pair type* $\mathsf{F}x : X.\ A[x]$. Its introduction form $\mathsf{F}\,(e, t)$ consists of a pair consisting of an intuitionistic term e of type X, and a linear term t of type $A[e]$ — observe the type of the linear term may depend on the intuitionistic first component. However, just as in the LNL calculus, this pair has a pattern matching eliminator $\mathsf{let}\ \mathsf{F}\,(x, a) = e\ \mathsf{in}\ e'$. Also, we add a dependent linear function space $\Pi x : X.\ A$, which is a function which takes an intuitionistic argument and returns a linear result (whose type may depend on the argument). We add this for essentially hygienic reasons — it gives us syntax for the equivalence $(\mathsf{F}x : X.\ A) \multimap C \simeq \Pi x : X.\ A \multimap C$.

The equality theory of our language is given in Figure 5. This gives the β and η rules for each of the connectives, including commuting conversions for all of the positive connectives (i.e., $\otimes$, I and F). We also include the equality reflection rule of extensional type theory. For space reasons, we do not include any of the congruence rules, or the equivalence relation axioms (they are given in the technical report).

$$\frac{}{\Gamma \vdash \mathsf{Loc} : \mathsf{U}_i} \qquad \frac{\Gamma \vdash e : \mathsf{Loc} \qquad \Gamma \vdash X : \mathsf{U}_i}{\Gamma \vdash e \mapsto X : \mathsf{L}_i} \qquad \frac{\Gamma \vdash A : \mathsf{L}_i}{\Gamma \vdash \mathsf{T}(A) : \mathsf{L}_i}$$

$$\frac{\Gamma \vdash A : \mathsf{L}_i}{\Gamma \vdash [A] : \mathsf{L}_i}$$

$$\frac{\Gamma \vdash X : \mathsf{U}_i \qquad \Gamma, x : X \vdash Y : \mathsf{U}_i}{\Gamma \vdash \forall x : X.\ Y : \mathsf{U}_i} \qquad \frac{\Gamma \vdash X : \mathsf{U}_i \qquad \Gamma, x : X \vdash Y : \mathsf{L}_i}{\Gamma \vdash \forall x : X.\ Y : \mathsf{L}_i}$$

$$\frac{\Gamma \vdash X : \mathsf{U}_i \qquad \Gamma, x : X \vdash Y : \mathsf{U}_i}{\Gamma \vdash \exists x : X.\ Y : \mathsf{U}_i} \qquad \frac{\Gamma \vdash X : \mathsf{U}_i \qquad \Gamma, x : X \vdash Y : \mathsf{L}_i}{\Gamma \vdash \exists x : X.\ Y : \mathsf{L}_i}$$

$$\frac{}{\Gamma \vdash \top_I : \mathsf{U}_i}$$

Figure 6. Well-formedness of extensions

$$\frac{\Gamma, x : X \vdash e : Y \qquad x \notin \mathrm{FV}(e)}{\Gamma \vdash e : \forall x : X.\ Y} \qquad \frac{\Gamma \vdash e : \forall x : X.\ Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e : [e'/x]Y}$$

$$\frac{\Gamma, x : X, y : Y \vdash e : Z \qquad x \notin \mathrm{FV}(e)}{\Gamma, y : \exists x : X.\ Y \vdash e : Z}$$

$$\frac{\Gamma, x : X \vdash Y\ \mathsf{type} \qquad \Gamma \vdash e' : X \qquad \Gamma \vdash e : [e'/x]Y}{\Gamma \vdash e : \exists x : X.\ Y}$$

$$\frac{\Gamma, x : X; \Delta \vdash e : A \qquad x \notin \mathrm{FV}(e)}{\Gamma; \Delta \vdash e : \forall x : X.\ A}$$

$$\frac{\Gamma; \Delta \vdash e : \forall x : X.\ A \qquad \Gamma \vdash e' : X}{\Gamma; \Delta \vdash e : [e'/x]A}$$

$$\frac{\Gamma, x : X; \Delta, a : A \vdash e : C \qquad x \notin \mathrm{FV}(e)}{\Gamma; \Delta, a : \exists x : X.\ A \vdash e : C}$$

$$\frac{\Gamma, x : X \vdash Y\ \mathsf{linear} \qquad \Gamma \vdash e' : X \qquad \Gamma; \Delta \vdash e : [e'/x]Y}{\Gamma; \Delta \vdash e : \exists x : X.\ Y}$$

$$\frac{\begin{array}{c}\Gamma, n : \mathbb{N} \vdash \Pi x : X[n].\ Y[n]\ \mathsf{type} \qquad \Gamma, f : \top_I, x : X(0) \vdash e : Y(0) \\ \Gamma, n : \mathbb{N}, f : \Pi x : X[n].\ Y[n], x : X[\mathsf{s}(n)] \vdash e : Y[\mathsf{s}(n)] \\ n \notin \mathrm{FV}(\mathsf{fix}\ f\ x = e)\end{array}}{\Gamma \vdash \mathsf{fix}\ f\ x = e : \forall n : \mathbb{N}.\ \Pi x : X[n].\ Y[n]}$$

Figure 7. Intersection and Union Types

3. Internal Imperative Programming

There have been a number of very successful applications of dependent type theory to verify imperative programs using dependent types, such as XCAP, Bedrock and the Verified Software Toolchain [4, 11, 31]. These approaches all have the common feature that they are "external" methodologies for imperative programming. That is, within the metalogic of dependent type theory, an imperative language (such as C or assembly) and its semantics is defined, and a program logic for that language is specified and proved correct. So the type theory remains pure, and imperative programs are therefore objects of study rather than programs of the type theory.

We would like to investigate an internal, "proofs as programs", methodology for imperative programming, where we write functional programs which (a) have effects, but (b) nonetheless are inhabitants of the appropriate types.

Pointers, Computations, and Proof Irrelevance In order to program with mutable state, we need types to describe both state and computation that manipulate it. To do this, we will follow the pattern of L3 [2], and introduce a type of *locations*, and a type of *reference capabilities*.

$$\frac{\Gamma \vdash e : \mathsf{Loc} \qquad \Gamma; \Delta \vdash t : [e \mapsto X] \qquad \Gamma, x : X; \Delta', a : [e \mapsto X] \vdash t' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ (x, a) = \mathsf{get}(e, t)\ \mathsf{in}\ t' : C}$$

$$\frac{\Gamma; \Delta \vdash e : A}{\Gamma; \Delta \vdash \mathsf{val}\, e : \mathsf{T}(A)} \qquad \frac{\Gamma; \Delta \vdash e : \mathsf{T}(A) \qquad \Gamma; \Delta', a : A \vdash e' : \mathsf{T}(C)}{\Gamma; \Delta, \Delta' \vdash \mathsf{let\ val}\ a = e\ \mathsf{in}\ e' : \mathsf{T}(C)}$$

$$\frac{\Gamma \vdash e : X}{\Gamma; \cdot \vdash \mathsf{new}_X\, e : \mathsf{T}((\mathsf{F}x : \mathsf{Loc}.\ [x \mapsto X]))}$$

$$\frac{\Gamma \vdash e : \mathsf{Loc} \qquad \Gamma; \Delta \vdash t : [e \mapsto X]}{\Gamma; \Delta \vdash \mathsf{free}\,(e, t) : \mathsf{T}(I)}$$

$$\frac{\Gamma \vdash e : \mathsf{Loc} \qquad \Gamma; \Delta \vdash t : [e \mapsto X] \qquad \Gamma \vdash e' : Y}{\Gamma; \Delta \vdash e :=_t e' : \mathsf{T}([e \mapsto Y])}$$

$$\frac{\Gamma; \Delta \vdash e \div A}{\Gamma; \Delta \vdash * : [A]} \qquad \frac{\Gamma; \Delta \vdash e : A}{\Gamma; \Delta \vdash e \div A}$$

$$\frac{\Gamma; \Delta \vdash e : [A] \qquad \Gamma; \Delta', x : A \vdash e' \div C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ [x] = e\ \mathsf{in}\ e' \div C}$$

$$\frac{\Gamma; \Delta \vdash e : [I] \qquad \Gamma; \Delta' \vdash e' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ [] = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Gamma; \Delta \vdash e : [A \otimes B] \qquad \Gamma; \Delta', a : [A], b : [B] \vdash e' : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ [a, b] = e\ \mathsf{in}\ e' : C}$$

Figure 8. Typing of Imperative Programs

$$\frac{\Gamma, x : X \vdash e \equiv e' : Y}{\Gamma \vdash e \equiv e' : \forall x : X.\ Y} \qquad \frac{\Gamma \vdash e \equiv e' : \forall x : X.\ Y \qquad \Gamma \vdash t : X}{\Gamma \vdash e \equiv e' : [t/x]Y}$$

$$\frac{\Gamma \vdash e \equiv e' : [t/x]Y \qquad \Gamma \vdash t : X}{\Gamma \vdash e \equiv e' : \exists x : X.\ Y}$$

$$\frac{\Gamma, x : X, y : Y \vdash e \equiv e' : Z \qquad x \notin \mathrm{FV}(e, e', Z)}{\Gamma, y : \exists x : X.\ Y \vdash e \equiv e' : Z}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let\ val}\, x = \mathsf{val}\, t\ \mathsf{in}\ t' \equiv [t/x]t' : \mathsf{T}(C)}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{let\ val}\, x = t\ \mathsf{in\ val}\, x \equiv t : \mathsf{T}(C)}$$

$$\frac{}{\Gamma; \Delta \vdash \begin{array}{c}\mathsf{let\ val}\, y = (\mathsf{let\ val}\, x = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \\ \equiv \\ \mathsf{let\ val}\, x = t_1\ \mathsf{in\ let\ val}\, y = t_2\ \mathsf{in}\ t_3\end{array} : \mathsf{T}(C)}$$

$$\frac{}{\Gamma; \Delta \vdash e \equiv e' : [A]}$$

$$\frac{\Gamma, x : X; \Delta \vdash e \equiv e' : A}{\Gamma; \Delta \vdash e \equiv e' : \forall x : X.\ Y} \qquad \frac{\Gamma; \Delta \vdash e \equiv e' : \forall x : X.\ A \qquad \Gamma \vdash t : X}{\Gamma; \Delta \vdash e \equiv e' : [t/x]A}$$

$$\frac{\Gamma; \Delta \vdash e \equiv e' : [t/x]A \qquad \Gamma \vdash t : X}{\Gamma; \Delta \vdash e \equiv e' : \exists x : X.\ A}$$

$$\frac{\Gamma, x : X; \Delta, a : A \vdash e \equiv e' : C \qquad x \notin \mathrm{FV}(e, e', C)}{\Gamma \vdash \Delta, a : \exists x : X.\ A \equiv e : e'C}$$

$$\frac{}{\Gamma \vdash (\mathsf{fix}\ f\ x = e)\ e' \equiv [(\mathsf{fix}\ f\ x = e)/f, e'/x]e : Z}$$

Figure 9. Imperative Equality

The type Loc is an intuitionistic type of locations. This is a type of addresses, which may be freely duplicated and stored in data structures. Testing locations for equality is a pure computation, and for simplicity we do not consider address arithmetic. However, merely having a location is not sufficient to dereference it, since *a priori* we do not know if that location points to allocated memory.

The right to access a pointer is encoded in the capability type $e \mapsto X$. This says that the location e holds a value of type X. This differs slightly from separation logic, whose points-to assertion $e \mapsto v$ says that e points to the specific value v. However, since we have dependent types at our disposal, we can easily recover the separation logic assertion with the encoding $e \mapsto v \equiv e \mapsto (\Sigma x : X.\ x = v)$.

At this point, much of the descriptive power of separation logic is already available to us: as soon as we have locations and pointer capabilities, we can use the tensor product $P \otimes Q$ just like the separating conjunction $P * Q$.[1]

Next, we introduce a type $\mathsf{T}(A)$ of computations. That is, if A is a linear type, then $\mathsf{T}(A)$ is a computation type which produces a result of type A. Note that $\mathsf{T}(A)$ is itself a linear type; intuitively, executing a computation modifies the current state so that it becomes a post-state of type A. The reason we treat computations monadically is because, naively, allocation and deallocation of memory violates linearity: for example, if we typed allocation as $\mathsf{F}x : \mathsf{Loc}.\ x \mapsto X$, the points-to assertion is "created from nothing". Since one of our design goals is precisely to keep the semantics (see section 4) as naive as possible, we introduce a linear computation type.

With the features we have introduced so far, a stateful computation e might be given the type $\mathsf{G}(P \multimap \mathsf{T}(Q))$, representing a linearly-closed term which takes a pre-state of type P, and computes a result of type Q. While this is perfectly adequate from a props-as-types point of view, it is not quite right computationally.

In our approach, we encode reasoning about state using linear types, so that we can read the type $P' \multimap P$ as saying that the state P' implies the state P. However, a proof f of this entailment is a linear lambda term, and evaluating it *does computation*. So if we precompose the proof f with the computation e, then we get a result which does different computation than e did originally. This is quite different from the way that the rule of consequence works in Hoare logic.

To solve this problem, we introduce a form of linear proof irrelevance into our language. The type $[A]$ is a *squash type*, which erases all of the proof-relevant data about A — we are left with only knowledge of its inhabitation. Our type system has a rule saying that $*$ is an inhabitant of $[A]$, if we can find evidence for it using the monadic judgement $\Gamma; \Delta \vdash e \div A$. This judgement lets us turn any derivation of A as evidence of inhabitation, and lets us use evidence of proof-irrelevant inhabitation relevantly, as long as we stay in the irrelevance judgement. We also give a pair of rules to distribute proof-irrelevance through tensor products and units.[2]

So we can type computations as $\mathsf{G}([P] \multimap \mathsf{T}(\mathsf{F}x : X.\ [Q]))$, representing computations that take (proof-irrelevant) pre-states P to proof-irrelevant post-states, while returning values of type X.

We can now describe the typing rules for computations in Figure 8. Given a location and a proof-irrelevant capability to access it, we can now type a dereference operation. Dereferencing $\mathsf{let}\ (x, a) = \mathsf{get}(e, t)\ \mathsf{in}\ t'$ (in Figure 8) takes a location e and a capability t, and then binds the contents to x, and the returned capability in the variable a, before evaluating the body t'.

The dereference operation is a pure linear term (i.e., has a non-monadic type), since it reads but does not modify the state. The other stateful operations are monadic. Allocation $\mathsf{new}_X\ e$ is a computation which returns a pair of location and an irrelevant capability to access that location. Deallocation $\mathsf{free}(e, e')$ takes a location and the capability to access it, and then deallocates the memory, returning a computation of unit type. Update $e :=_t e'$ takes a location and a capability to access it, and then updates the location, returning a new capability to access it. Since capabilities are linear, we are able to support *strong update* – assignment can change the type of a pointer. We also include standard monadic rules for returning pure values and sequencing computations.

As a simple example, consider the following term:

```
1  let val F (x, c1) = new_ℕ 0 in
2  let val F (y, c2) = new_bool false in
3  let (n, c1') = get(x, c1) in
4  let val c2' = (y :=_c2 n) in
5  val (F (x, c1'), F (y, c2'))
```

On the first two lines, this function allocates two pointers x (pointing to 0) and y (pointing to false), with access capabilities c_1 and c_2. On line 3, it dereferences x, binding the result to n. Furthermore, the linear capability c_1 is taken by the dereference operation, and returned (as the capability c_1'). On line 4, y is updated to point to n, again taking in the capability c_2 and returning the capability c_1. This is a strong update, changing the type of y so that it points to a natural number. Then, we return both pointers and their access capabilities on line 5, with an overall type of $\mathsf{T}((\mathsf{F}x : \mathsf{Loc}.\ [x \mapsto \mathbb{N}]) \otimes (\mathsf{F}y : \mathsf{Loc}.\ [y \mapsto \mathbb{N}]))$.

One point worth mentioning is that our computation monad is a *strong* monad, which is in some sense the essence of the frame rule of separation logic — the derivability of the map $(A \otimes \mathsf{T}(B)) \multimap \mathsf{T}((A \otimes B))$ means that a computation of B that doesn't use the state A can "fold it in" to produce a bigger computation of $A \otimes B$.

Another point worth mentioning is that our system only treats reading pointers as a pure linear action. Naively, one might expect writes could also be viewed as pure, since linearity ensures that there is at most one capability to modify a location. Unfortunately, this idea is incompatible with proof-irrelevance: squashing erases a term, and erasing terms which could modify the heap would be unsound.

Intersection Types and Implicit Quantification At this point, we are close to being able to formulate specifications of imperative programs in an internal way. However, one still-missing feature of program logics is support of *logical variables*. These are variables which occur only in specifications, but which do not affect the evaluation of a program. Fortunately, these have also been studied extensively in lambda-calculus, albeit under a different name: *intersection types*.

We introduce the two type formers $\forall x : X.\ Y$ and $\exists x : X.\ Y$ (and their linear counterparts), corresponding to an X-indexed

[1] It is relatively easy to show that any formula in the logic of bunched implications [33] which does not interleave the ordinary implication $P \supset Q$ and the magic wand $P -\!\!* Q$ has a corresponding proof in linear logic, since interleaving the additive and multiplicative implications is the only way to create bunched contexts. However, the magic wand is rarely used in practice (though see [21]).

[2] Intuitionistically, the isomorphism $[A \times B] \simeq [A] \times [B]$ is derivable, but the proof relies on the fact that pairing has first and second projections. With the linear tensor product, this isomorphism has to be built into the syntax.

intersection and union, respectively. A term e is an inhabitant of $\forall x : X.\ Y$ if it is an inhabitant of Y for all x, and symmetrically, e inhabits $\exists x : X.\ Y$ if there is some $v : X$ such that $e : [v/x]Y$. The precise rules are given in Figure 7 – the key constraint is that even though a new hypothesis becomes available in the context, it cannot occur in the term. This ensures that it cannot affect evaluation, even though it can appear in typing.

The classical intersection type $A \wedge B$ can be encoded using large elimination, as $\forall b : 2.\ \mathsf{if}(b, A, B)$, and the union type $A \vee B$ is $\exists b : 2.\ \mathsf{if}(b, A, B)$.

Now, we can introduce logical variables into our specifications (i.e., types) by simple quantification:

$$\forall x : X.\ \mathsf{G}\,([P] \multimap \mathsf{T}\,(\mathsf{F} y : Y.\ [Q]))$$

The variable x can now range over both the pre- and the post-conditions without being able to affect the computation.

Fixed Points Finally, we make use of intersection types to introduce a recursive function form $\mathsf{fix}\ f\ x = e$ (see Figure 7). This function is ascribed the type $\forall n : \mathbb{N}.\ \Pi x : X[n].\ Y[n]$, when (1) assuming f has the type $\Pi x : X[n].\ Y[n]$ and x has the type $X[\mathsf{s}(n)]$, then the body has the type $Y[\mathsf{s}(n)]$, and (2) we can show that the body has the type $Y[0]$ when the argument has type $X[0]$. (In the base case, we also assume that f has a top type which cannot be used, to prevent recursive calls.)

This permits us to define recursive functions if we can supply a decreasing termination metric as a computationally-irrelevant argument. This is useful because the inductive eliminators require a proof-relevant argument, which we might not have!

3.1 Examples

With all of this machinery, we can now define a simple linked list predicate. For simplicity, we leave the data out, leaving only the link structure behind.

$$\begin{array}{lcl} \mathsf{LList} & : & \mathbb{N} \times \mathsf{Loc} \to \mathsf{L}_0 \\ \mathsf{LList}(0, l) & = & [l \mapsto \mathsf{inl}\,()] \\ \mathsf{LList}(n+1, l) & = & \mathsf{F} l' : \mathsf{Loc}.\ [l \mapsto \mathsf{inr}\, l'] \otimes \mathsf{LList}(n, l') \end{array}$$

This says that a list of length 0 is a pointer to a null value, and a list of length $n+1$ is a pointer to a list of length n. Note that this predicate is *proof-relevant*, in that a term of type $\mathsf{LList}(n, l)$ is a nested tuple including all the pointers in the linked list. As a result, we need to squash the predicate when giving specifications to programs. We will also make use of the fact that proof-relevant quantifiers inside of a squash can be turned into proof-irrelevant quantifiers outside of a squash: that is, the axiom $[\mathsf{F} x : X.\ A] \multimap \exists x : X.\ [A]$ is inhabited (by the term $\lambda x.\ *$). In particular, this lets us turn squashed coproducts into union types: $[A \oplus B] \multimap [A] \vee [B]$.

List length As a first example, if we wanted to define a length function for linked lists, we would need to do it as:

$$\begin{array}{l} \mathsf{len} : \forall n : \mathbb{N}.\ \Pi l : \mathsf{Loc}.\ \mathsf{G}([\mathsf{LList}(n, l)] \\ \qquad\qquad \multimap (\mathsf{F} m : \mathbb{N}, \mathsf{pf} : m = n.\ [\mathsf{LList}(n, l)])) \\ \mathsf{len}\ l = \mathsf{G}(\lambda c. \\ \quad \mathsf{let}\ [c'] = * \ \mathsf{in} \\ \quad \mathsf{let}\ [c_1, c_2] = c' \ \mathsf{in} \\ \quad \mathsf{let}\ ((v, \mathsf{pf}), c_1') = \mathsf{get}(l, c_1)\ \mathsf{in} \\ \quad \mathsf{case}(v, \\ \qquad \mathsf{inl}\,() \to \mathsf{F}\,(0, \mathsf{refl}, *) \\ \qquad \mathsf{inr}\, l' \to \mathsf{let}\ (\mathsf{F}\,(m, \mathsf{pf}, c'')) = \mathsf{G}^{-1}\,(\mathsf{len}\ l') * \mathsf{in} \\ \qquad\qquad \mathsf{F}\,(m+1, \mathsf{refl}, *))) \end{array}$$

The type of this function says that if we are given a list of length n, then we can recursively traverse the linked list in order to get a computationally-relevant number equal to n. This turns out to be a very common pattern in specifications: we use computations to mediate between some logical fact about a piece of data (i.e., the index n) and computational data (in this case, the length m of the list).

Obviously, this function cannot be defined with an eliminator on n, since the point is precisely to compute a relevant number equal to n, and the data we receive is merely a bare pointer, along with the right to traverse it.

Worth noting is that the squash type completely conceals almost all of the proof-related book-keeping related to manipulating the linked-list predicate LList. As a result, it is obvious what the computationally-relevant action is, though *why* things work can get quite tricky to figure out. In this case, on the first line, we making use of the fact that given a list $[\mathsf{LList}(n, l)]$ we can get out an element of the union type $\exists p : n = 0.\ [l \mapsto \mathsf{inl}\,() \otimes l] \vee \exists k : \mathbb{N}, n = \mathsf{s}(k), l' : \mathsf{Loc}.\ [l \mapsto \mathsf{inr}\, l' \otimes \mathsf{LList}(k, l')]$. This is bound to c', and then the body can be typechecked under both alternatives.

List Append We can also write a type-theoretic version of in-place list append, one of the examples from the original paper on separation logic. First, let's write a rehead function, which changes the head pointer of a list:

$$\begin{array}{l} \mathsf{rehead} : \forall n : \mathbb{N}.\ \Pi l, l' : \mathsf{Loc}.\ \mathsf{G}([l \mapsto \mathsf{inl}\,()] \otimes [\mathsf{LList}(n, l')] \\ \qquad\qquad \multimap \mathsf{T}\,([\mathsf{LList}(n, l)])) \\ \mathsf{rehead}\ l\ l' = \mathsf{G}(\lambda c_1\ c_2. \\ \quad \mathsf{let}\ [c_2', c_2''] = * \ \mathsf{in} \\ \quad \mathsf{let}\ (v, c_2') = \mathsf{get}(l', c_2')\ \mathsf{in} \\ \quad \mathsf{let}\ \mathsf{val}\, c_1 = (l :=_{c_1} v)\ \mathsf{in} \\ \quad \mathsf{let}\ \mathsf{val}\,() = \mathsf{free}\,(l', c_2')\ \mathsf{in} \\ \quad \mathsf{val}\, *) \end{array}$$

This function simply assigns the contents the list to the first argument pointer, and then frees the head pointer of the second list. As usual, a small amount of bureaucracy is needed to split irrelevant arguments, but otherwise the code is very similar to what one might write in ML or Haskell.

$$\begin{array}{l} \mathsf{append} : \forall n, n' : \mathbb{N}.\ \Pi l, l' : \mathsf{Loc}.\ \mathsf{G}([[\mathsf{LList}(n, l)] \otimes [\mathsf{LList}(n', l')] \\ \qquad\qquad \multimap \mathsf{T}\,([\mathsf{LList}(n + n', l)])) \\ \mathsf{append}\ l\ l' = \mathsf{G}(\lambda c_1\ c_2. \\ \quad \mathsf{let}\ [c_1', c_1''] = * \ \mathsf{in} \\ \quad \mathsf{let}\ (v, c_1') = \mathsf{get}(l, c_1')\ \mathsf{in} \\ \quad \mathsf{case}(v, \\ \qquad \mathsf{inl}\,() \to \mathsf{let}\ \mathsf{val}\, c = \mathsf{G}^{-1}(\mathsf{rehead}\ l\ l')\ c_1'\ c_2\ \mathsf{in}\ \mathsf{val}\, * \\ \qquad \mathsf{inr}\, l'' \to \mathsf{let}\ \mathsf{val}\, c = \mathsf{G}^{-1}(\mathsf{append}\ l''\ l')\ c_1''\ c_2\ \mathsf{in}\ \mathsf{val}\, *)) \end{array}$$

This walks to the end of a list, and then reheads the head of the second list to the tail of the first list. Then with some straightforward linearly-typed proof-programming, it's possible to show that this gives us a linked list of the appropriate shape (which is concealed in the $\mathsf{val}\, *$ terms). The fact that we return only a squashed argument corresponds to the fact that this is an in-place algorithm, which modifies the state and returns only some linear knowledge about the new state.

Data Abstraction Because we can quantify over types, our calculus also supports data abstraction, in the style of Nanevski et al. [29]. As an example, we can use our linked list to implement a stack as an abstract type. First, we define a function to grow a linked list by one element.

$$\mathsf{grow} : \Pi l : \mathsf{Loc}.\ \forall n : \mathbb{N}.\ \mathsf{G}(\mathsf{LList}(l, n) \multimap \mathsf{T}\,(\mathsf{LList}(l, n+1)))$$

$$
\begin{array}{l}
\mathsf{grow}\ l = \mathsf{G}(\lambda c.\\
\ \ \mathsf{let}\ [c', c''] = * \ \mathsf{in}\\
\ \ \mathsf{let}\ (v, c') = \mathsf{get}(l, c')\ \mathsf{in}\\
\ \ \mathsf{case}(v,\\
\ \ \ \ \mathsf{inl}\,() \to \mathsf{let\ val}\,(l_0, c_0) = \mathsf{new\ inl}\,()\ \mathsf{in}\\
\qquad\qquad \mathsf{let\ val}\,c' = l :=_{c'} \mathsf{inr}\,l_0\ \mathsf{in}\\
\qquad\qquad \mathsf{val}\,*\\
\ \ \ \ \mathsf{inr}\,l'' \to \mathsf{let\ val}\,c = \mathsf{G}^{-1}(\mathsf{grow}\ l'')\ c''\ \mathsf{in\ val}\,*))
\end{array}
$$

This iterates to the end of the list, and then adds one element to the tail, in order to preserve the identity of the list. Similarly, we can shrink a list by an element as well.

$$
\begin{array}{l}
\mathsf{shrink} : \Pi l : \mathsf{Loc}.\ \forall n : \mathbb{N}.\ \mathsf{G}(\mathsf{LList}(l, n+1) \multimap \mathsf{T}\,(\mathsf{LList}(l, n)))\\
\mathsf{shrink}\ l = \mathsf{G}(\lambda c.\\
\ \ \mathsf{let}\ [c', c''] = * \ \mathsf{in}\\
\ \ \mathsf{let}\ (l', c') = \mathsf{get}(l, c')\ \mathsf{in}\\
\ \ \mathsf{let\ val}\,c' = l :=_{c'} \mathsf{inl}\,()\ \mathsf{in}\\
\ \ \mathsf{G}^{-1}(\mathsf{rehead}\ l\ l')\ c'\ c'')
\end{array}
$$

This deletes an element of the list, using the rehead function to ensure that the head pointer remains unchanged.

Now, we can give a procedure to build a stack as an abstract type.

$$
\begin{array}{l}
\mathsf{make} : \mathsf{G}(\mathsf{T}(\mathsf{F\ Stack} : \mathbb{N} \to \mathsf{L}_i,\\
\qquad\qquad \mathsf{push} : \forall n : \mathbb{N}.\ \mathsf{G\,Stack}\ n \multimap \mathsf{T}\,(\mathsf{Stack}(\mathsf{s}(n))),\\
\qquad\qquad \mathsf{pop} : \forall n : \mathbb{N}.\ \mathsf{G\,Stack}(\mathsf{s}(n)) \multimap \mathsf{T}\,(\mathsf{Stack}\ n).\\
\qquad\qquad \mathsf{Stack}\ 0))\\
\mathsf{make} = \mathsf{G}(\mathsf{T}(\mathsf{let\ val\,F}\,(l, c) = \mathsf{new}\ 0\ \mathsf{in}\\
\qquad\qquad \mathsf{val\,F}\,(\lambda n.\ \mathsf{LList}(n, l), \mathsf{F}\,(\mathsf{grow}\ l, \mathsf{F}\,(\mathsf{shrink}\ l, c)))))
\end{array}
$$

This procedure allocates a new location initialized to 0, and then returns the grow and shrink operations at the types for push and pop. Importantly, the representation of the stack is completely hidden — there is no way to know a pointer represents the stack.

One particularly interesting feature of this specification is that the choice of representation invariant is *data-dependent*: our definition of the Stack predicate captures the actual location allocated as a free variable.

3.2 Equational Theory

So far, we have decomposed the Hoare triple $\{P\}\,x : X\,\{Q\}$ as the dependent linear type $\mathsf{G}\,([P] \multimap \mathsf{T}\,((\mathsf{F}x : X.\ [Q])))$. That is, we view elements of Hoare type as duplicable terms, which take a proof-irrelevant witness for the pre-state P, and then compute a result of type X, as well as a producing a proof-irrelevant witness for the post-state Q.

This is, admittedly, a zoo of connectives! However, such fine-grained decompositions have a number of benefits. Since each of the connectives we give has a very simple $\beta\eta$-theory, we can derive the equational theory for terms of Hoare type by joining up the $\beta\eta$-theory of its components.[3]

As a simple example, an immediate consequence of our decomposition is that all terms of type $\{0\}\,x : X\,\{Q\}$ (that is, the type of computations with a false precondition) are equal, simply via the eta-rule for functions and falsity.

Another consequence (which we rely upon in our examples) comes from our encoding of the points-to predicate of separation logic. Our basic pointer type $l \mapsto X$ says only that l contains a value of type X. We encode the points-to of separation logic $l \mapsto v$ using the type $l \mapsto \Sigma x : X.\ x = v$. Then, the fact that the sigma-type is contractible (i.e., all its elements are equal) means that we know that any dereference of l is equal to (v, refl), without having to put explicit proofs into the terms (because of equality reflection).

In addition, moving things like preconditions out of the computation monad $\mathsf{T}\,(A)$ lets us give a very simple model for it, which means that we can easily show the soundness of equations such as eliminating duplicate gets

$$
\Gamma; \Delta \vdash \begin{array}{c} \mathsf{let}\ (x, c) = \mathsf{get}(e, e')\ \mathsf{in}\ C[\mathsf{let}\ (x', c') = \mathsf{get}(e, c)\ \mathsf{in}\ e''] \\ \equiv \\ \mathsf{let}\ (x, c) = \mathsf{get}(e, e')\ \mathsf{in}\ C[[c/c', x/x']e''] \end{array} : A
$$

and eliminating gets after sets:

$$
\Gamma; \Delta \vdash \begin{array}{c} \mathsf{let\ val}\,c = e :=_{e''} e'\ \mathsf{in}\ C[\mathsf{let}\ (x', c') = \mathsf{get}(e, c)\ \mathsf{in}\ e'''] \\ \equiv \\ \mathsf{let\ val}\,c = e :=_{e''} e'\ \mathsf{in}\ C[[e'/x', c/c']e'''] \end{array} : \mathsf{T}\,(A)
$$

3.3 Hoare Type Theory

Hoare type theory [28] gives an alternative internal approach to imperative programming, by *directly* embedding Hoare triples into type theory. The central idea is to use dependent types to specify imperative programs as elements of an indexed monad, where the index domain are formulas of separation logic. This approach has proven remarkably successful in verifying complex imperative programs (such as modern congruence closure algorithms [30]), but working out the equational theory of the Hoare type has proven to be an extremely challenging problem, to the point that Svendsen et al. [40] give a model without any equations.

To illustrate the connection more clearly, we note that each of the basic constants of Hoare type theory is definable in our calculus, with a type that is close to the image of the translation of its Hoare type. Below, we give allocation, dereference, assignment, and the rule of consequence to illustrate. (We leave out the terms, because they can be read off the types.) In each case, the Hoare type is on the top line, and the linear type is on the line below.

$$
\begin{array}{ll}
\mathsf{new}_1 : & \{\mathsf{emp}\}\,x : \mathsf{Loc}\,\{x \mapsto ()\}\\
& \mathsf{G}\,(\mathsf{T}\,((\mathsf{F}x : \mathsf{Loc}.\ [x \mapsto ()])))\\
\mathsf{set} : & \Pi x : \mathsf{Loc}, v, v' : X.\ \{x \mapsto v\}1\{x \mapsto v'\}\\
& \forall v : X.\ \Pi x : \mathsf{Loc}, v' : X.\ \mathsf{G}\,([x \mapsto v] \multimap \mathsf{T}\,([x \mapsto v']))\\
\mathsf{get} : & \Pi x : \mathsf{Loc}, v : X.\ \{x \mapsto v\}v' : X\{v' = v \wedge x \mapsto v\}\\
& \forall v : X.\ \Pi x : \mathsf{Loc}.\ \mathsf{G}\,([x \mapsto v] \multimap \mathsf{F}v' : X, \mathsf{pf} : v' = v.\ [x \mapsto v])\\
\mathsf{con} : & \Pi P, P', Q, Q'.\ \begin{array}{l}(P \supset P') \to (Q' \supset Q) \to\\ \{P'\}x : X\{Q'\} \to\\ \{P\}x : X\{Q\}\end{array}\\
& \Pi P, P', Q, Q' : \mathsf{L}_i.\ \begin{array}{l}\mathsf{G}\,[P \multimap P'] \to \mathsf{G}\,[Q' \multimap Q] \to\\ \mathsf{G}\,([P'] \multimap \mathsf{F}x : X.\ [Q']) \to\\ \mathsf{G}\,([P] \multimap \mathsf{F}x : X.\ [Q])\end{array}
\end{array}
$$

We can see that in some cases, the LNL_D typings have computational advantages. For example, for the set constant, we are able to use the implicit $\forall$ quantifier to make clear that while the pointer and the new value need to be passed as arguments, the old contents do not need to be passed in. This is not possible in HTT (due to the absence of intersections), and to work around this a two-state postcondition is used. As another example, the get constant doesn't have a monadic effect in LNL_D.

[3] We say "the" equational theory because the $\beta\eta$-equality is canonical. However, there are many desirable equations (such as those arising from parametricity) which go far beyond it.

As a result, any (totally correct) program in Hoare type theory should in principle be translatable into $\mathrm{LNL_D}$ (neglecting the much superior maturity and automation available for Hoare type theory).

Caveat There is, however, a significant difference between $\mathrm{LNL_D}$ and HTT: the latter supports general recursion, and deals with partial correctness. $\mathrm{LNL_D}$ requires a termination metric on fixed point definitions and only deals with total correctness.

Most of the complexity in the model theory of HTT comes from its treatment of recursion, such as the need for $\top\top$-closure to ensure admissibility for fixed point induction. We have sidestepped these issues by focussing only on total correctness.

In fact, much of the subtlety in the design of $\mathrm{LNL_D}$ lay in the effort needed to *rule out* general recursion — in languages with higher-order store, it is usually possible to write recursion operators by backpatching function pointers (aka "Landin's knot"), and so we had to carefully exploit linearity to prevent the typability of such examples. (This was originally observed by Ahmed et al. [2].) Another example is found in our fixed point operator, which defines recursive *functions* with an irrelevant termination metric. If we had given a general fixed point operator $\mathsf{fix}\ x.\ e$, even with an irrelevant termination metric, then looping computations would be definable. (This is connected to the ability of intersection types to detect evaluation order.)

As a result, to formally argue that our approach makes proving equations easier, we would need to give a version of our $\mathsf{T}(-)$ monad which is partial, which would give a calculus corresponding more closely to HTT. We hope to investigate these questions more deeply in future work. As matters stand, all we can say is that our semantics is in many ways much less technically sophisticated than models such as the one of Petersen et al. [34], even though it permits us to prove a number of novel equations on programs.

4. Metatheory

We now establish the metatheory of our type system. Our approach integrates the ideas of Harper [20] and Ahmed et al. [2]. We begin by giving an untyped operational semantics for the language. In fact, we give three operational semantics: one for pure intuitionistic terms, one for pure linear terms, and one for monadic linear terms.

Next, we use this operational semantics to give a semantic realizability model, by defining the interpretation of each logical connective as a partial equivalence relation (PER). Because we have dependent types, we need to define the set of syntactic types mutually recursively with the interpretation of each type, in the style of an inductive-recursive definition. Furthermore, we have to interpret intuitionistic and linear terms differently: we use an ordinary PER on terms to interpret intuitionistic types, and a PER on machine configurations (terms plus store) to interpret linear types.

This gives a semantic interpretation of closed types. We extend this to an interpretation of contexts as sets of environments, and then prove the fundamental theorem of logical relations for our type system, which establishes the soundness and adequacy of our rules (including its equational theory).

4.1 Operational Semantics

The syntax for our language of untyped terms is given in Figure 10. For simplicity, all our types and terms live in the same syntactic category, regardless of whether they are intuitionistic and linear. The variable v ranges over intuitionistic values, u ranges over linear values, and σ denotes heaps.

There are actually *three* operational semantics for our language. First, we have an evaluation relation $e \Downarrow v$ for intuitionistic terms, a a linear evaluation relation $\langle \sigma; e\rangle \Downarrow \langle \sigma'; u\rangle$ for linear terms, and a monadic evaluation relation $\langle \sigma; e\rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\, u\rangle$ for evaluating monadic terms. Both the linear and monadic semantics are store-passing relations, because they are imperative computations which may depend upon the state of memory.

The intuitionistic evaluation relation $e \Downarrow v$ is a standard call-by-name big-step evaluation relation for the lambda calculus. We treat all types as values (and do not evaluate their subterms), and also include location values l in the syntactic category of values. Though there are no typing rules for location values, they may arise during program evaluation when fresh memory cells are allocated.

The linear evaluation relation $\langle \sigma; e\rangle \Downarrow \langle \sigma'; u\rangle$ evaluates the machine configuration $\langle \sigma; e\rangle$ to $\langle \sigma'; u\rangle$. This is also a call-by-name big-step evaluator. However, there are many rules in this reduction relation, simply because we have many linear types in our language. The rules for functions (both linear $A \multimap B$ and dependent $\Pi x : X.\ A$) and pairs $A \,\&\, B$ follow the beta-rules of the equational theory, as expected. The let-style eliminators for I, $A \otimes B$ and $\mathsf{F}x : X.\ A$ are more verbose, but also line up with the beta-rules of the theory. The eliminator $\mathsf{G}^{-1}\, e$ evaluates e to a term $\mathsf{G}\, t$, and then executes t, again following the beta-rule.

A nuisance in our rules is that we need operational rules for the let-style rules (i.e., $\mathsf{let}\ [a, b] = e\ \mathsf{in}\ e'$) distributing proof irrelevance through units and tensors. This has no computational content, since the argument e is never evaluated, but nevertheless we must include evaluation rules for it.

Finally, we also have the dereference operation $\mathsf{let}\ (x, c') = \mathsf{get}(e, e')\ \mathsf{in}\ e''$. This rule evaluates e to a location l, and e' to an irrelevant value $*$. Then it looks up the value v stored at l, and binds that to x, and binds c' to the irrelevant value $*$ before evaluating the continuation e''.

One point worth making is that *none* of these rules affect the shape of the heap – we can dereference pointers in the pure linear fragment, but we cannot write or modify them. As a result, every linear reduction is really of the form $\langle \sigma; e\rangle \Downarrow \langle \sigma; u\rangle$. We could have easily have made this into a syntactic restriction, but having two configurations is slightly more convenient when giving the realizability semantics.

Finally, we have the monadic evaluation $\langle \sigma; e\rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\, u\rangle$. The rule for let-binding $\mathsf{let}\ \mathsf{val}\, x = e\ \mathsf{in}\ e'$ is sequential composition: it evaluates the first argument to a monadic value, and then binds the result to x before evaluating e'. The rule for $\mathsf{new}\ e$ evaluates its (intuitionistic) argument to a value v, and then allocates a fresh location l to store it in, returning the location (and a dummy $*$ for the capability). The rule for assignments $e :=_{e''} e'$ evaluates e to a location l, e' to a value v, and e'' to a capability $*$. Then it modifies the heap so that l points to v. Similarly, the rule for deallocation $\mathsf{free}\,(e, e')$ evaluates e to l and then deallocates the pointer. Note that all of these operations could get stuck, if the appropriate piece of heap did not exist — the soundness proof will establish that type-safe programs are indeed safe.

4.2 CPPOs and Fixed Points

Since types depend on terms in dependent type theory, we cannot define the grammar of types up front, before giving an interpretation their interpretations. Instead, what we need is morally an inductive-recursive definition. As a result, we

$$
\begin{array}{lcl}
e, t, X, A & ::= & \Pi x : X.\ Y \mid A \multimap B \mid \lambda x : C.\ e \mid e\ e' \mid \hat{\lambda} x.\ e \\
& \mid & 1 \mid \mathsf{I} \mid () \mid \mathsf{let}\,() = e\ \mathsf{in}\ e' \\
& \mid & \Sigma x : X.\ Y \mid A \otimes B \mid (e, e') \\
& \mid & \pi_1\, e \mid \pi_2\, e \mid \mathsf{let}\,(x, y) = e\ \mathsf{in}\ e' \\
& \mid & \mathsf{G}\, e \mid \mathsf{G}^{-1}\, e \\
& \mid & \mathsf{F} x : X.\ A \mid \mathsf{F}\,(e, t) \mid \mathsf{let}\,\mathsf{F}\,(x, a) = t\ \mathsf{in}\ t' \\
& \mid & \top \mid A \,\&\, B \\
& \mid & \forall x : X.\ Y \mid \exists x : x.\ Y \\
& \mid & e =_X e' \mid \mathsf{refl} \\
& \mid & \mathbb{N} \mid 0 \mid \mathsf{s}(e) \mid \mathsf{iter}(e, 0 \to e_0, \mathsf{s}(x), y \to e_1) \\
& \mid & U_i \mid L_i \\
& \mid & x \mid \mathsf{fix}\, f\, x = e \\
& \mid & [A] \mid \mathsf{let}\,[x] = e\ \mathsf{in}\ e \mid * \\
& \mid & e \mapsto X \mid \mathsf{Loc} \mid \mathsf{new}_X\, e \mid \mathsf{free}\,(e, t) \mid l \\
& \mid & \mathsf{let}\,(x, a) = \mathsf{get}(e, e')\ \mathsf{in}\ e'' \mid e :=_{e''} e' \\
& \mid & \mathsf{T}(A) \mid \mathsf{val}\, e \mid \mathsf{let}\,\mathsf{val}\, x = e\ \mathsf{in}\ e' \\
\\
v & ::= & \lambda x : A.\ e \mid () \mid (e, e) \mid \mathsf{refl} \mid \mathsf{G}\, e \mid l \mid * \mid 0 \mid \mathsf{s}(v) \\
& \mid & \Pi x : X.\ Y \mid A \multimap B \mid \top \mid A \,\&\, B \\
& \mid & 1 \mid \mathsf{I} \mid \Sigma x : X.\ Y \mid A \otimes B \mid \mathsf{F} x : X.\ B \\
& \mid & e =_X e' \mid e \mapsto X \mid \mathsf{Loc} \mid U_i \mid L_i \\
\\
u & ::= & \lambda x.\ e \mid () \mid (e, e) \mid \mathsf{F}\,(e, e) \mid \hat{\lambda} x.\ e \mid (e, e') \\
& \mid & * \mid \mathsf{val}\, e \mid \mathsf{let}\,\mathsf{val}\, x = e\ \mathsf{in}\ e \mid \mathsf{new}_X\, e \mid e :=_{e''} e' \\
\\
\sigma & ::= & \cdot \mid \sigma, l : v
\end{array}
$$

Figure 10. Terms e, t, X, A, values v, linear values u, stores σ

need to use a fixed theorem which is a little stronger than the standard Kleene or Knaster-Tarski fixed point theorems.

Recall that a *pointed partial order* is a triple $(X, \leqslant, \bot)$ such that X is a set, $\leqslant$ is a partial order on X, and $\bot$ is the least element of X. A subset $D \subseteq X$ is a *directed set* when every pair of elements $x, y \in D$ has an upper bound in D (i.e., there is a $z \in D$ such that $x \leqslant z$ and $y \leqslant z$). A pointed partial order is *complete* (i.e., forms a CPPO) when every directed set D has a supremum $\bigsqcup D$ in X.

The following fixed point theorem[4] is in Harper [20], and is Theorem 8.22 in Davey and Priestley [13].

Theorem 1. *(Fixed Points on CPPOs) If X is a CPPO, and $f : X \to X$ is a monotone function on X, then f has a least fixed point.*

Proof. Construct the ordinal-indexed sequence x_α, where:

$$
\begin{array}{lcl}
x_0 & = & \bot \\
x_{\beta+1} & = & f(x_\beta) \\
x_\lambda & = & \bigsqcup_{\beta<\lambda} x_\beta
\end{array}
$$

Because f is monotone, we can show by transfinite induction that every initial segment is directed, which ensures the needed suprema exist and the sequence is well-defined.

Now, we know there must be a stage λ such that $x_\lambda = x_{\lambda+1}$. If there were not, then we could construct a bijection between the ordinals and the strictly increasing chain of elements of the sequence x. However, the elements of the sequence x are all drawn from X. Since X is a set, it follows that the elements of x must themselves form a set. Since the ordinals do not form a set (they are a proper class), this leads to a contradiction. Hence, f must have a fixed point. □

4.3 Partial Equivalence Relations and Semantic Type Systems

We now need to define what our types are, and how to interpret them. Following the example of Nuprl, we will

[4] A constructive, albeit impredicative, proof of this theorem is possible: this is Pataraia's theorem [17].

$$\boxed{e \Downarrow v}$$

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow \lambda x : A.\ e \qquad [e_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow (e_1, e_2) \qquad e_1 \Downarrow v}{\pi_1\, e \Downarrow v} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_2 \Downarrow v}{\pi_2\, e \Downarrow v}$$

$$\frac{e \Downarrow v}{\mathsf{s}(e) \Downarrow \mathsf{s}(v)} \qquad \frac{e \Downarrow 0 \qquad e_0 \Downarrow v}{\mathsf{iter}(e, 0 \to e_0, \mathsf{s}(x), y \to e_1) \Downarrow v}$$

$$\frac{e \Downarrow \mathsf{s}(n) \qquad \mathsf{iter}(n, 0 \to e_0, \mathsf{s}(x), y \to e_1) \Downarrow v \qquad [n/x, v/y]e_1 \Downarrow v'}{\mathsf{iter}(e, 0 \to e_0, \mathsf{s}(x), y \to e_1) \Downarrow v'}$$

$$\frac{e \Downarrow \mathsf{fix}\, f\, x = e_0 \qquad [(\mathsf{fix}\, f\, x = e_0)/f, e'/x]e_0 \Downarrow v}{e\ e' \Downarrow v}$$

$$\boxed{\langle \sigma; e \rangle \Downarrow \langle \sigma'; u \rangle}$$

$$\frac{}{\langle \sigma; u \rangle \Downarrow \langle \sigma; u \rangle}$$

$$\frac{\langle \sigma; e_1 \rangle \Downarrow \langle \sigma'; \lambda x.\ e_1' \rangle \qquad \langle \sigma'; [e_2/x]e_1' \rangle \Downarrow \langle \sigma''; u'' \rangle}{\langle \sigma; e_1\ e_2 \rangle \Downarrow \langle \sigma''; u'' \rangle}$$

$$\frac{\langle \sigma; e_1 \rangle \Downarrow \langle \sigma'; \hat{\lambda} x.\ e \rangle \qquad \langle \sigma'; [e_2/x]e \rangle \Downarrow \langle \sigma''; u'' \rangle}{\langle \sigma; e_1\ e_2 \rangle \Downarrow \langle \sigma''; u'' \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; (e_1, e_2) \rangle \qquad \langle \sigma'; e_1 \rangle \Downarrow \langle \sigma''; u'' \rangle}{\langle \sigma; \pi_1\, e \rangle \Downarrow \langle \sigma''; u'' \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; (e_1, e_2) \rangle \qquad \langle \sigma'; e_2 \rangle \Downarrow \langle \sigma''; u'' \rangle}{\langle \sigma; \pi_2\, e \rangle \Downarrow \langle \sigma''; u'' \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; () \rangle \qquad \langle \sigma'; e' \rangle \Downarrow \langle \sigma''; u \rangle}{\langle \sigma; \mathsf{let}\,() = e\ \mathsf{in}\ e' \rangle \Downarrow \langle \sigma''; u \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; (e_1, e_2) \rangle \qquad \langle \sigma'; [e_1/a, e_2/b]e' \rangle \Downarrow \langle \sigma''; u \rangle}{\langle \sigma; \mathsf{let}\,(a, b) = e\ \mathsf{in}\ e' \rangle \Downarrow \langle \sigma''; u \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; \mathsf{F}\,(e_1, e_2) \rangle \qquad \langle \sigma'; [e_1/x, e_2/a]e' \rangle \Downarrow \langle \sigma''; u \rangle}{\langle \sigma; \mathsf{let}\,\mathsf{F}\,(x, a) = e\ \mathsf{in}\ e' \rangle \Downarrow \langle \sigma''; u \rangle}$$

$$\frac{e \Downarrow \mathsf{G}\, e' \qquad \langle \sigma; e' \rangle \Downarrow \langle \sigma'; u \rangle}{\langle \sigma; \mathsf{G}^{-1}\, e \rangle \Downarrow \langle \sigma'; u \rangle} \qquad \frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; u' \rangle}{\langle \sigma; \mathsf{let}\,[] = e_0\ \mathsf{in}\ e \rangle \Downarrow \langle \sigma'; u' \rangle}$$

$$\frac{\langle \sigma; [*/a, */b]e \rangle \Downarrow \langle \sigma'; u' \rangle}{\langle \sigma; \mathsf{let}\,[a, b] = e_0\ \mathsf{in}\ e \rangle \Downarrow \langle \sigma'; u' \rangle}$$

$$\frac{e \Downarrow l \qquad \langle \sigma; e' \rangle \Downarrow \langle \sigma', l : v; * \rangle \qquad \langle \sigma', l : v; [v/x, */c]e'' \rangle \Downarrow \langle \sigma''; u \rangle}{\langle \sigma; \mathsf{let}\,(x, c) = \mathsf{get}(e, e')\ \mathsf{in}\ e'' \rangle \Downarrow \langle \sigma''; u \rangle}$$

$$\boxed{\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\, v \rangle}$$

$$\frac{}{\langle \sigma; \mathsf{val}\, e \rangle \rightsquigarrow \langle \sigma; \mathsf{val}\, e \rangle}$$

$$\frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; u \rangle \qquad e \neq \mathsf{val}\, u_0 \qquad \langle \sigma; u \rangle \rightsquigarrow \langle \sigma''; \mathsf{val}\, u' \rangle}{\langle \sigma; e \rangle \rightsquigarrow \langle \sigma''; \mathsf{val}\, u' \rangle}$$

$$\frac{\langle \sigma; e \rangle \rightsquigarrow \langle \sigma_1; \mathsf{val}\, e_1 \rangle \qquad \langle \sigma_1; [e_1/x]e' \rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\, v \rangle}{\langle \sigma; \mathsf{let}\,\mathsf{val}\, x = e\ \mathsf{in}\ e' \rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\, v \rangle}$$

$$\frac{e \Downarrow v \qquad l \notin \mathrm{dom}(\sigma)}{\langle \sigma; \mathsf{new}_X\, e \rangle \rightsquigarrow \langle \sigma, l : v; \mathsf{val}\,\mathsf{F}\,(l, *) \rangle}$$

$$\frac{e \Downarrow l \qquad e' \Downarrow v \qquad \langle \sigma; e'' \rangle \Downarrow \langle \sigma', l : v'; * \rangle}{\langle \sigma, l : v'; e :=_{e''} e' \rangle \rightsquigarrow \langle \sigma', l : v; \mathsf{val}\, * \rangle}$$

$$\frac{e \Downarrow l \qquad \langle \sigma; t \rangle \Downarrow \langle \sigma', l : v; * \rangle}{\langle \sigma; \mathsf{free}\,(e, t) \rangle \rightsquigarrow \langle \sigma'; \mathsf{val}\,() \rangle}$$

Figure 11. Operational Semantics

interpret types as partial equivalence relations of terms, with the intuition that if $(e, e) \in X$, then e is an element of the type X, and that if $(e, e') \in X$ then e is equal to e'. Because of type dependency, we will then *simultaneously* define the PER of types, and an interpretation function sending each type to the PER it defines.

A *partial equivalence relation (*PER*)* is a symmetric, transitive relation on closed, terminating expressions. We further require that PERs be *closed under evaluation*. Given a PER R, we require that for all e, e', v, v' such that $e \Downarrow v$ and $e' \Downarrow v'$, we have that $(e, e') \in R$ if and only if $(v, v') \in R$. Given a PER P, we write P^* to close it up under evaluation. We extend this notation to functions, as well, so that given a function f into PERs, $f^*(x) = (f\ x)^*$. Finally, given a relation R, we write $R^\dagger$ to take its symmetric, transitive closure. We will use PERs to interpret intuitionistic types.

A *partial evaluation relation on configurations (*CPER*)* is a symmetric, transitive relation on terminating machine configurations $\langle \sigma; e \rangle$. We further require that they be *closed under linear evaluation*. Given a CPER M, we require that for all $\langle \sigma_1; e_1 \rangle$ such that $\langle \sigma_1; e_1 \rangle \Downarrow \langle \sigma_1'; u_1 \rangle$ and $\langle \sigma_2; e_2 \rangle$ such that $\langle \sigma_2; e_2 \rangle \Downarrow \langle \sigma_2'; u_2 \rangle$, we have $(\langle \sigma_1; e_1 \rangle, \langle \sigma_2; e_2 \rangle) \in M$ if and only if $(\langle \sigma_1'; u_1 \rangle, \langle \sigma_2'; u_2 \rangle) \in M$. We will use CPERs to interpret linear types, with the idea that the store components of each element of the PER corresponds to the resources owned by that term.

Note that since evaluation (both ordinary and linear) is deterministic, an evaluation-closed PER is determined by its sub-PER on values (or value configurations). As a result, we will often define PERs and functions on PERs by simply giving the relation for values.

A *semantic linear/non-linear type system* is a four-tuple $(I \in \text{PER}, L \in \text{PER}, \phi : |I| \to \text{PER}, \psi : |L| \to \text{CPER})$ such that ϕ respects I and ψ respects L. We say that I are the *semantic intuitionistic types*, L are the *semantic linear types*, and ϕ and ψ are *the type interpretation functions*.

The set of type systems forms a CPPO. The least element is the type system $(\emptyset, \emptyset, !_{\text{PER}}, !_{\text{CPER}})$ with an empty set of intuitionistic and linear types. The ordering $(I, L, \phi, \psi) \leqslant (I', L', \phi', \psi')$ is given by set inclusion on $I \subseteq I'$ and $L \subseteq L'$, when there is agreement between ϕ and ϕ' on the common part of their domains, and likewise for ψ and ψ' (which we write $\phi \sqsubseteq \phi'$ and $\psi \sqsubseteq \psi'$). Given a directed set, the join is given by taking unions pointwise (treating the functions ϕ and ψ as graphs).

Next, our goal is to define a semantic type system to interpret our syntax. To do this, we will define a monotone type system operator T_k. Since type systems form a CPPO, we can define the semantic type system we use to interpret our syntax as the least fixed point of T_k.

First, we'll define some constructions on PERs in Figures 12 and 13. There is one construction for each of the type connectives in our language, explaining what each connective means "logical relations style" — though these constructions do not yet define a logical relation, since they are just constructions on PERs.

The intuitionistic constructions are given in Figure 12. The *Loc* relation relates each location to itself, and the unit relation $\hat{1}$ relates the unit value to itself. The identity relation $Id(a, b, E)$ is equal to $\{(\mathsf{refl}, \mathsf{refl})\}$ if a and b are in the relation E, and is empty otherwise. The Π construction takes a PER E, and a function Φ from elements of E to PERs (which respects the relation E). From that data it returns the PER of function values (f, f') which take E-related pairs (a, a') to $(f\ a, f'\ a')$ in $E(a)$, with the $\Sigma(E, \Phi)$ construction working similarly for pairs. The intersection type $\hat{\forall}(E, \Phi)$ construction just takes the intersection over $\Phi(a)$ for all a in E, which is a PER since PERs are closed under intersection. The union type is a little more complicated; it takes the union of PERs, and then has to take the symmetric transitive closure because PERs are not closed under unions. The natural number relation $\hat{\mathbb{N}}$ relates numerals to themselves, and $\hat{\top}_I$ relates any two values.

Loc	$=$	$\{(l, l) \mid l \in \mathsf{Loc}\}$
$\hat{1}$	$=$	$\{((), ())\}$
$Id(a, b, E)$	$=$	$\{(\mathsf{refl}, \mathsf{refl}) \mid (a, b) \in E\}$
$\Pi(E, \Phi)$	$=$	$\{(v, v') \mid \forall (a, a') \in E.\ (v\,a, v'\,a') \in \Phi(a)\}$
$\Sigma(E, \Phi)$	$=$	$\{((a, b), (a', b')) \mid (a, a') \in E \wedge (b, b') \in \Phi(a)\}$
$G(C)$	$=$	$\{(\mathsf{G}\,e, \mathsf{G}\,e') \mid (\langle \cdot; e \rangle, \langle \cdot; e' \rangle) \in C\}$
$\hat{\forall}(E, \Phi)$	$=$	$\{(v, v') \mid \forall (e, e') \in E.\ (v, v') \in \Phi(e)\}$
$\hat{\exists}(E, \Phi)$	$=$	$\{(v, v') \mid \exists (e, e') \in E.\ (v, v') \in \Phi(e)\}^\dagger$
$\hat{\mathbb{N}}$	$=$	$\{(\mathsf{s}^k(0), \mathsf{s}^k(0)) \mid k \text{ is a natural number}\}$
$\hat{\top}_I$	$=$	$\{(v, v') \mid v \in \text{Val} \wedge v' \in \text{Val}\}$

Figure 12. Intuitionistic PER constructions

The linear constructions are given in Figure 13. The $\hat{\top}$ relation relates two unit values under any stores at all, whereas the $\hat{1}$ relation relates two units only in the empty store. Similarly, the $A \mathbin{\&} B$ relation relates two pairs, as long as the first components are related at A using all of the store, and the second components are related at B using all of the store. The tensor relation $C \mathbin{\hat{\otimes}} D$ relation, on the other hand, relates two pairs if their stores can divided into two disjoint parts, one relating the first components in the C relation, and the other relating the second components in the D relation. The semantic linear function space constructor $C \mathbin{\hat{\multimap}} D$ relates functions, such that if we have values in C with disjoint store, then the applications are related in D with the combined store (much like the "magic wand" of separation logic). The $F(E, \Psi)$ relation consists of pairs of stores and values $\mathsf{F}(a, b)$ and $\mathsf{F}(a', b')$, such that a and a' are intuitionistic terms related in E, and b and b' are related in $\Psi(a)$ using their whole stores. The Π_L relation works similarly, except that it relates on dependent linear functions rather than pairs. The linear intersection and union type constructors work similarly to their intuitionistic counterparts.

The proof irrelevance relation $Irr(A)$ relates two squash tokens $*$ at some stores, if there are linear terms that could relate those stores at A. The pointer construction $Ptr(e, E)$ simply says that the heap consists of a single memory cell addressed by e, and whose contents are related at E. (Its only term is $*$, because we don't care about its value, as we use it solely to track capabilities.) The $\hat{\mathsf{T}}(A)$ construction relates monadic computations, if each side reduces to a store/value configuration which is related by A. Furthermore, only "frame-respecting" computations can be related — if a computation works in a smaller store, it should also work in a larger store, without modifying that larger store.

We can now define an operator T_k on type systems:

$$T_k(I, L, \phi, \psi) = (I'^*, L'^*, \phi'^*, \psi'^*)$$

where I' and L' are defined in Figure 14 and and where ϕ' and ψ' are defined in Figure 15.

The definition of I' and L' includes each of the base types, and then for each of the higher-arity operations draws the argument types from I or L, so that (for example) L' includes every $A \otimes B$ where A and B are in L. The definition of ϕ' and ψ' is by analysis of the structure of the elements of I' and L', using the PER constructions we described earlier

$$
\begin{array}{lcl}
\hat{\top} & = & \{(\langle\sigma;()\rangle,\langle\sigma';()\rangle) \mid \sigma,\sigma' \in \mathrm{Store}\} \\
A \mathbin{\hat{\&}} B & = & \left\{ \begin{array}{l|l} (\langle\sigma;(a,b)\rangle, & (\langle\sigma;a\rangle,\langle\sigma';a'\rangle) \in A \wedge \\ \langle\sigma';(a',b')\rangle) & (\langle\sigma;b\rangle,\langle\sigma';b'\rangle) \in B \end{array} \right\} \\
\hat{I} & = & \{(\langle\cdot;()\rangle,\langle\cdot;()\rangle)\} \\
(C \mathbin{\hat{\otimes}} D) & = & \left\{ \begin{array}{l|l} (\langle\sigma;(c,d)\rangle, & \exists \sigma_C, \sigma_D, \sigma_{C'}, \sigma_{D'}. \\ \langle\sigma';(c',d')\rangle) & \sigma = \sigma_C \cdot \sigma_D \wedge \\ & \sigma' = \sigma'_C \cdot \sigma'_D \wedge \\ & (\langle\sigma_C;c\rangle,\langle\sigma'_C;c'\rangle) \in C \wedge \\ & (\langle\sigma_D;d\rangle,\langle\sigma'_D;d'\rangle) \in D \end{array} \right\} \\
(C \mathbin{\hat{\multimap}} D) & = & \left\{ \begin{array}{l|l} (\langle\sigma;u\rangle, & \forall \sigma_0 \# \sigma, \sigma'_0 \# \sigma', c, c'. \\ \langle\sigma';u'\rangle) & \text{if } (\langle\sigma_0;c\rangle,\langle\sigma'_0;c'\rangle) \in C \\ & \text{then } \left(\begin{array}{l} \langle\sigma \cdot \sigma_0; u c\rangle, \\ \langle\sigma' \cdot \sigma'_0; u' c'\rangle \end{array} \right) \in D \end{array} \right\} \\
F(E,\Psi) & = & \left\{ \begin{array}{l|l} (\langle\sigma;F(a,b)\rangle, & E(a,a') \wedge \\ \langle\sigma';F(a',b')\rangle) & (\langle\sigma;b\rangle,\langle\sigma';b'\rangle) \in \Psi(a) \end{array} \right\} \\
\Pi_L(E,\Psi) & = & \left\{ \begin{array}{l|l} (\langle\sigma;u\rangle, & \forall (e,e') \in E. \\ \langle\sigma';u'\rangle) & (\langle\sigma;u\,e\rangle,\langle\sigma';u'\,e'\rangle) \in \Psi(e) \end{array} \right\} \\
\hat{\forall}_L(E,\Psi) & = & \left\{ \begin{array}{l|l} (\langle\sigma;u\rangle, & \forall (e,e') \in E. \\ \langle\sigma';u'\rangle) & (\langle\sigma;u\rangle,\langle\sigma';u'\rangle) \in \Psi(e) \end{array} \right\} \\
\hat{\exists}_L(E,\Psi) & = & \left\{ \begin{array}{l|l} (\langle\sigma;u\rangle, & \exists (e,e') \in E. \\ \langle\sigma';u'\rangle) & (\langle\sigma;u\rangle,\langle\sigma';u'\rangle) \in \Psi(e) \end{array} \right\}^{\dagger} \\
\hat{T}(A) & = & \left\{ \begin{array}{l|l} (\langle\sigma_1;e_1\rangle, & \forall \sigma_f \# \sigma_1, \sigma_g \# \sigma_2.\ \exists \sigma'_1, \sigma'_2, u_1, u_2. \\ \langle\sigma_2;e_2\rangle) & \langle\sigma_1 \cdot \sigma_f; e_1\rangle \rightsquigarrow \langle\sigma'_1 \cdot \sigma_f; \mathsf{val}\, u_1\rangle \wedge \\ & \langle\sigma_2 \cdot \sigma_g; e_2\rangle \rightsquigarrow \langle\sigma'_2 \cdot \sigma_g; \mathsf{val}\, u_2\rangle \wedge \\ & (\langle\sigma'_1;u_1\rangle,\langle\sigma'_2;u_2\rangle) \in A \end{array} \right\} \\
Ptr(e,E) & = & \left\{ (\langle\sigma_1;*\rangle,\langle\sigma_2;*\rangle) \left| \begin{array}{l} \sigma_1 = [l : v_1] \wedge \sigma_2 = [l : v_2] \wedge \\ (e,l) \in Loc \wedge (v_1,v_2) \in E \end{array} \right. \right\} \\
Irr(A) & = & \{(\langle\sigma;*\rangle,\langle\sigma';*\rangle) \mid \exists a, a'.\ (\langle\sigma;a\rangle,\langle\sigma;a'\rangle) \in A\ \}
\end{array}
$$

Figure 13. Linear PER Constructions

to interpret each of the type constructors. Furthermore, in the definition of ϕ', when we write $\phi'(\mathsf{Loc})$, we actually mean any $\phi'(e)$ such that $I'(e,\mathsf{Loc})$. This is justified by the fact that evaluation is deterministic and relations are closed under evalution. (Similarly, we define ψ' only on the values, extending it to all terms by determinacy and closure under evaluation.) Once we have this definition, we can then show that T_k is a monotone type system operator.

Lemma 1 (T_k is monotone)**.** *We have that T_k is a monotone function on type systems.*

Proof. (Sketch) This lemma really makes two claims. First, that T_k is indeed a function on semantic type systems (in particular, that ϕ' and ψ' respect I' and L'), and second, that it is monotone. Both of these follow from a case analysis on the shape of the possible elements of I' and L'. See the technical report for details. □

Then, we can take the interpretation of the i-th universe $\mathcal{T}_i$ to be the least fixed point of $\overline{T}_i$. Furthermore, we also have a cumulativity property:

Lemma 2 (Universe Cumulativity)**.** *If $i \leqslant k$ then $\mathcal{T}_i \leqslant \mathcal{T}_k$.*

We sometimes write $\mathcal{T}$ for the limit of the countable universe hierarchy, and write U, L, ϕ, and ψ for its components.

4.4 Semantic Environments

Our semantic type system $\mathcal{T}$ gives us an interpretation of closed types. Before we can prove the correctness of our typing rules, we have to extend our interpretation to open types and terms, and to do that, we have to give an interpretation of environments.

In Figure 16, we give the interpretation of contexts. The meaning of an intuitionistic context $[\![\Gamma]\!]$ is given as a set of binary substitutions γ. By binary substitution, we mean that for every $x \in \mathrm{dom}(\gamma)$, we have that $\gamma(x) = (e,e')$ for some

$$
\begin{array}{lcl}
I' = & & \{(\mathsf{Loc},\mathsf{Loc})\} \cup \\
& & \{(\mathbb{N},\mathbb{N})\} \cup \\
& & \{(\top_I,\top_I)\} \cup \\
& & \{(e_1 =_X e_2, t_1 =_Y t_2) \mid I(X,Y) \wedge \Phi(X)(e_1,t_1) \wedge \Phi(X)(e_2,t_2)\} \cup \\
& & \left\{ \begin{array}{l|l} (\Pi x : X.\ Y[x], & I(X,X') \wedge \\ \Pi x : X'.\ Y'[x]) & \forall (v,v') \in \Phi(X).\ I(Y[v],Y'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\Sigma x : X.\ Y[x] & I(X,X') \wedge \\ \Sigma x : X'.\ Y'[x]) & \forall (v,v') \in \Phi(X).\ I(Y[v],Y'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\forall x : X.\ Y[x] & I(X,X') \wedge \\ \forall x : X'.\ Y'[x]) & \forall (v,v') \in \Phi(X).\ I(Y[v],Y'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\exists x : X.\ Y[x] & I(X,X') \wedge \\ \exists x : X'.\ Y'[x]) & \forall (v,v') \in \Phi(X).\ I(Y[v],Y'[v']) \end{array} \right\} \cup \\
& & \{(G\,A, G\,A') \mid L(A,A')\} \cup \\
& & \{(U_i,U_i) \mid i < k\} \cup \\
& & \{(L_i,L_i) \mid i < k\}
\end{array}
$$

$$
\begin{array}{lcl}
L' = & & \{(I,I)\} \cup \\
& & \{(A \otimes B, A' \otimes B') \mid L(A,A') \wedge L(B,B')\} \cup \\
& & \{(A \multimap B, A' \multimap B') \mid L(A,A') \wedge L(B,B')\} \cup \\
& & \left\{ \begin{array}{l|l} (\mathsf{F}x : X.\ A[x], & I(X,X') \wedge \\ \mathsf{F}x : X'.\ A'[x]) & \forall (v,v') \in \Phi(X).\ L(A[v],A'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\Pi x : X.\ A[x], & I(X,X') \wedge \\ \Pi x : X'.\ A'[x]) & \forall (v,v') \in \Phi(X).\ L(A[v],A'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\forall x : X.\ A[x], & I(X,X') \wedge \\ \forall x : X'.\ A'[x]) & \forall (v,v') \in \Phi(X).\ L(A[v],A'[v']) \end{array} \right\} \cup \\
& & \left\{ \begin{array}{l|l} (\exists x : X.\ A[x], & I(X,X') \wedge \\ \exists x : X'.\ A'[x]) & \forall (v,v') \in \Phi(X).\ L(A[v],A'[v']) \end{array} \right\} \cup \\
& & \{(\top,\top)\} \cup \\
& & \{(A \mathbin{\&} B, A' \mathbin{\&} B') \mid L(A,A') \wedge L(B,B')\} \cup \\
& & \{(\mathsf{T}(A),\mathsf{T}(A')) \mid (A,A') \in L\} \cup \\
& & \{(e \mapsto X, e' \mapsto X') \mid (e,e') \in \mathsf{Loc} \wedge (X,X') \in I\}
\end{array}
$$

Figure 14. Definition of type part of T_k

$$
\begin{array}{lcl}
\phi'(\mathsf{Loc}) & = & \mathit{Loc} \\
\phi'(\mathbb{N}) & = & \hat{\mathbb{N}} \\
\phi'(\top_I) & = & \hat{\top}_I \\
\phi'(e_1 =_X e_2) & = & \mathit{Id}(e_1,e_2,\phi(X)) \\
\phi'(\Pi x : X.\ Y[x]) & = & \Pi(\phi(X),\lambda v.\ \phi(Y[v])) \\
\phi'(\Sigma x : X.\ Y[x]) & = & \Sigma(\phi(X),\lambda v.\ \phi(Y[v])) \\
\phi'(\forall x : X.\ Y[x]) & = & \hat{\forall}(\phi(X),\lambda v.\ \phi(Y[v])) \\
\phi'(\exists x : X.\ Y[x]) & = & \hat{\exists}(\phi(X),\lambda v.\ \phi(Y[v])) \\
\phi'(G\,A) & = & G(\psi(A)) \\
\phi'(U_i) \text{ when } i < k & = & \mathsf{let}\ (\hat{U},\hat{L},\hat{\phi},\hat{\psi}) = \mathrm{fix}(T_i)\ \mathsf{in}\ \hat{U} \\
\phi'(L_i) \text{ when } i < k & = & \mathsf{let}\ (\hat{U},\hat{L},\hat{\phi},\hat{\psi}) = \mathrm{fix}(T_i)\ \mathsf{in}\ \hat{L} \\
\\
\psi'(I) & = & \hat{I} \\
\psi'(A \otimes B) & = & \psi(A) \mathbin{\hat{\otimes}} \psi(B) \\
\psi'(\top) & = & \hat{\top} \\
\psi'(A \mathbin{\&} B) & = & \psi(A) \mathbin{\hat{\&}} \psi(B) \\
\psi'(A \multimap B) & = & \psi(A) \mathbin{\hat{\multimap}} \psi(B) \\
\psi'(\mathsf{F}x : X.\ A[x]) & = & F(\phi(X),\lambda v.\ \psi(A[v])) \\
\psi'(\Pi x : X.\ A[x]) & = & \Pi_L(\phi(X),\lambda v.\ \psi(A[v])) \\
\psi'(\forall x : X.\ A[x]) & = & \hat{\forall}_L(\phi(X),\lambda v.\ \psi(A[v])) \\
\psi'(\exists x : X.\ A[x]) & = & \hat{\exists}_L(\phi(X),\lambda v.\ \psi(A[v])) \\
\psi'(\mathsf{T}(A)) & = & T(\psi(A)) \\
\psi'(e \mapsto X) & = & \mathit{Ptr}(e,\phi(X))
\end{array}
$$

Figure 15. Definition of T_k, interpretation part

$$
\begin{array}{lcl}
[\![\cdot]\!] & = & \{\langle\rangle\} \\
[\![\Gamma, x : X]\!] & = & \left\{ (\gamma, (e_1, e_2)/x) \;\middle|\; \begin{array}{l} \gamma \in [\![\Gamma]\!] \wedge \\ (\gamma_1(X), \gamma_2(X)) \in U \wedge \\ (e_1, e_2) \in \phi(\gamma_1(X)) \end{array} \right\}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\cdot]\!] & = & \{(\langle \epsilon; \langle\rangle \rangle, \langle \epsilon; \langle\rangle \rangle)\} \\
[\![\Delta_1, \Delta_2]\!] & = & \left\{ \begin{array}{c} (\langle \sigma; \delta_1, \delta_2 \rangle, \\ \langle \sigma'; \delta_1', \delta_2' \rangle) \end{array} \;\middle|\; \begin{array}{l} \exists \sigma_1, \sigma_2, \sigma_1', \sigma_2'. \\ \sigma = \sigma_1 \cdot \sigma_2 \wedge \sigma' = \sigma_1' \cdot \sigma_2' \wedge \\ (\langle \sigma_1; \delta_1 \rangle, \langle \sigma_1'; \delta_1' \rangle) \in [\![\Delta_1]\!] \wedge \\ (\langle \sigma_2; \delta_2 \rangle, \langle \sigma_2'; \delta_2' \rangle) \in [\![\Delta_2]\!] \end{array} \right\} \\
[\![a : A]\!] & = & \left\{ (\langle \sigma; e/a \rangle, \langle \sigma'; e'/a \rangle) \;\middle|\; \begin{array}{l} (A, A) \in L \wedge \\ (\langle \sigma; e \rangle, \langle \sigma; e' \rangle) \in \psi(A) \end{array} \right\}
\end{array}
$$

Figure 16. Interpretation of Environments

pair of terms. We write γ_1 for the ordinary unary substitution substituting the first projection of γ for each variable in γ, and γ_2 for the ordinary substitution substituting the second projection. We write $\gamma(e)$ for the pair $(\gamma_1(e), \gamma_2(e))$.

The intepretation of the empty context is the empty substitution. The interpretation of $\Gamma, x : X$ is every substitution $(\gamma, (e, e')/x)$ where $\gamma \in [\![\Gamma]\!]$ and $\gamma_1(X)$ is a type in U, and (e, e') are related by $\phi(\gamma_1(X))$.

The interpretation of linear contexts Δ is a little more complicated. Instead of a binary substitution, its elements consist of pairs of store/substitution pairs. Intuitively, a linear environment is a set of linear values for each linear variable, plus the store needed to interpret the value. So the interpretation of a single-variable context $a : A$ are all the pairs of configurations $(\langle \sigma; e/a \rangle, \langle \sigma'; e'/a \rangle)$, where $\langle \sigma; e \rangle$ and $\langle \sigma'; e' \rangle$ are related by $\psi(A)$ (and A is a linear type in L).

The interpretation of the empty linear context is similar to the interpretation for I; it relates two empty substitutions in empty heaps. Likewise, the interpretation of contexts Δ_1, Δ_2 is very similar to the interpretation of the tensor product. When we interpret a context Δ_1, Δ_2, we take it be a pair of $\langle \sigma; \delta_1, \delta_2 \rangle$ and $\langle \sigma'; \delta_1', \delta_2' \rangle$, such that we can divide the stores so that $\sigma = \sigma_1 \cdot \sigma_2$ and $\sigma' = \sigma_1' \cdot \sigma_2'$, and $\langle \sigma_1; \delta_1 \rangle$ is related to $\langle \sigma_1'; \delta_1' \rangle$ at Δ_1, and likewise $\langle \sigma_2; \delta_2 \rangle$ is related to $\langle \sigma_2'; \delta_2' \rangle$ at Δ_2.

4.5 Fundamental Property

We can now state the fundamental lemma of logical relations:

Theorem 2 (Fundamental Property)**.**
Suppose Γ ok *and* $\gamma \in [\![\Gamma]\!]$ *and* $\Gamma \vdash \Delta$ ok *and* $(\langle \sigma_1; \delta_2 \rangle, \langle \sigma_1; \delta_2 \rangle) \in [\![\gamma_1(\Delta)]\!]$. *Then we have that:*

1. *If* $\Gamma \vdash X$ type *then* $\gamma(X) \in U$.
2. *If* $\Gamma \vdash X \equiv Y$ type *then* $(\gamma_1(X), \gamma_2(Y)) \in U$.
3. *If* $\Gamma \vdash e : X$ *then* $\gamma(e) \in \phi(\gamma_1(X))$.
4. *If* $\Gamma \vdash e_1 \equiv e_2 : X$ *then* $(\gamma_1(e_1), \gamma_2(e_2)) \in \phi(\gamma_1(X))$.
5. *If* $\Gamma \vdash A$ linear *then* $\gamma(A) \in L$.
6. *If* $\Gamma \vdash A \equiv B$ linear *then* $(\gamma_1(A), \gamma_2(B)) \in L$.
7. *If* $\Gamma; \Delta \vdash e : A$ *then* $(\langle \sigma_1; \gamma_1(\delta_1(e)) \rangle, \langle \sigma_2; \gamma_2(\delta_2(e)) \rangle) \in \psi(\gamma_1(X))$.
8. *If* $\Gamma; \Delta \vdash e_1 \equiv e_2 : A$ *then* $(\langle \sigma_1; \gamma_1(\delta_1(e_1)) \rangle, \langle \sigma_2; \gamma_2(\delta_2(e_2)) \rangle) \in \psi(\gamma_1(X))$.
9. *If* $\Gamma; \Delta \vdash e \div A$ *then there are* t *and* t' *such that for each* $\gamma \in [\![\Gamma]\!]$ *and* $(\langle \sigma_1; \delta_1 \rangle, \langle \sigma_2; \delta_2 \rangle) \in [\![\gamma_1(\Delta)]\!]$, *we have* $(\langle \sigma_1; \delta_1(\gamma_1(t)) \rangle, \langle \sigma_2; \delta_2(\gamma_2(t')) \rangle) \in \psi(\gamma_1(A))$.

Proof. The theorem follows by a massive mutual induction over all of the rules of all of the judgements in the system. The full proof is in the companion technical report.

Worth noting is that in many of the cases, we make the arbitrary choice to use the first projection of the substitution (i.e., γ_1 or δ_1) rather than the second. Connosieurs of dependent types may worry that difficulties could be lurking in this choice. Happily, there are no problems, because we established that $\mathcal{T}$ is a semantic type system up front, and so we know that ϕ and ψ respect I and U. □

Evaluation of all closed well-typed terms terminates. The fundamental property also implies consistency, since every well-typed term is in the logical relation and inconsistent types (like $0 = 1$) have empty relations. We also have adequacy — any two provably equal terms of natural number type will evaluate to the same numeral (and similarly for the linear type $\mathsf{Fn} : \mathbb{N}.\ I$ and the monadic type $T\ ((\mathsf{Fn} : \mathbb{N}.\ I))$).

5. Implementation

We have written a small type checker for a version of our theory in Ocaml. We say "version of", because the type system we present in this paper is (extremely) undecidable, and so we have needed to make a number of modifications to the type theory to make it implementable. We emphasize that even though all of the modifications we made seem straightforward, we have done no proofs about the implementation: we still need to show that everything is consistent and conservative with respect to the extensional system.

Equality Reflection Our implementation does not support equality reflection, or indeed most of the η rules. Our system implements an untyped conversion relation, and opportunistically implements a few of the easier eta-rules (for example, $\lambda x.\ f\ x \equiv f$). So our implementation is really an intensional type theory, rather than an extensional one.

Intersection and Union Types The formulation of intersections and unions in this paper is set up to optimize the simplicity of the side conditions — to check a term e against the type $\forall x : X.\ Y$, we just add $x : X$ to the context, and check that x does not occur free in e. However, eliminating an intersection type requires guessing the instantiation.

In our implementation, we have to write an explicit abstraction $\lambda x.\ e$ to introduce a universal quantifier $\forall x : X.\ Y$, and eliminating an intersection requires an explicit annotation, so that if $f : \forall x : X.\ Y$, and we want to instantiate it with $e : X$, then we need to write $f\ e : [e/x]Y$. Similarly, we give a syntax for implicit existentials $\exists x : X.\ Y$ where the introduction is an explicit pack $\mathsf{pack}(e, t)$, with $e : X$ and $t : [e/x]Y$, and the elimination is an unpack $\mathsf{let\ pack}(x, y) = e\ \mathsf{in}\ e'$.

However, we track which variables are computationally relevant or not: in both a forall-introduction $\lambda x.\ e$, and a exists-elimination $\mathsf{let\ pack}(x, y) = e\ \mathsf{in}\ e'$, the x is not computationally relevant, may only appear within type annotations, implicit applications and the first argument of packs. This restriction is in the style of the "resurrection operator" of Pfenning [35] (which was later extended to type dependency by Abel [1]). Our overall result also looks very much like ICC* [6], the variant of the implicit calculus of constructions [27] with decidable typechecking.

Proof Irrelevance In this paper, we implement linear proof irrelevance with a "squashed" term $*$. To turn this into something we can typecheck, we note that the premise of the rule requires that we produce an irrelevant derivation $\Gamma; \Delta \vdash e \div A$. The rules of this judgement (see Figure 8) are actually a linear version of the monadic modality of Pfenning and Davies [36] (and in fact we borrow their notation). So, to turn this into something we can typecheck, all we have to do is to replace $*$ with a term $[e]$, where we can derive $\Gamma; \Delta \vdash e \div A$.

We should also extend definitional equality to equate all terms of type $[A]$, though we have not yet implemented this, as that requires a typed conversion relation.

Evaluation Our preliminary experience is that equality is the most challenging issue for us. We had to postulate numerous equalities that are sound in our model, but not applied automatically, and this is quite noisy. For example, we needed to postulate function extensionality to make things work, since we formulate lists in terms of W-types [25], and they are not well-behaved intensionally. In addition, our by-hand derivations of uses $\mathsf{fix}\ f\ x = e$ often make use of equality reflection, but we have not yet found a completely satisfactory formulation of its typing rule that works well in an intensional setting. An easy fix is to add lists (and other inductive types) as a primitive to our system, and another possibility is to move to something like observational type theory [3].

6. Discussion and Related Work

We divide the discussion into several categories, even though many individual pieces of work straddle multiple categories.

Dependent Types As an extensional type theory, our system is constructed in the Nuprl [12] tradition. Our metatheory was designed following the paper of Harper [20], and many of our logical connectives were inspired by work on Nuprl. For example, our treatment of intersection and union types is similar to the work of Kopylov [23] (as well as PER models of polymorphism such as the work of Bainbridge et al. [5]). Our treatment of proof irrelevance is also inspired by the "squash types" of Nogin [32], though because of the constraints of linearity we could not directly use his rules.

Linearity Cervesato and Pfenning [10] proposed the *Linear Logical Framework*, a core type theory for integrating linearity and dependent types. This calculus consisted of the negative fragment of MALL ($\multimap$, $\&$, $\top$) together with a dependent function space $\Pi x : A.\ B$, with the restriction that in any application $f\ e$, the argument e could contain no free linear variables. In our calculus, this type corresponds to $\Pi x : \mathsf{G}\,A.\ B$. This neatly explains why the argument of the application should not have any free linear variables; every LLF application $f\ e$ corresponds to an application $f\ (\mathsf{G}\,e)$ in our calculus!

Very recently, Vákár [42] has proposed an extension of Linear LF, in which he extends linear LF with an intuitionistic sigma-type $\Sigma x : \widehat{A}.\ B$, whose introduction form is a pair whose first component does not have any free linear variables. In our calculus, this can be encoded as $\mathsf{F}x : \mathsf{G}\,A.\ B$. This encoding yields a nice explanation of why the first component of the pair is intuitionistic, and also why the dependent pair has a let-style elimination rather than a projective eliminator. (He also extends the calculus with many other features such as equality types and universes, just as we have, but in both cases these extensions are straightforward and so there is nothing interesting to compare.) Instead of an operational semantics, he gives a categorical axiomatization, and in future work we will see if his semantics applies to our syntax.

Polarization, Value Dependency and Proof Irrelevance Another extension of LLF is the *Concurrent Logical Framework* (or CLF) of Watkins et al. [43], which extends LLF with positive types (such as $A \oplus B$, $A \otimes B$, and the intuitionistic sigma types), as well as a *computational monad* $\{A\}$ to mediate between uses of positive and negative types. This seems to be an instance of *focalization*, in which a pair of adjoint modalities $\uparrow N$ and $\downarrow P$ are used to segregate positive and negative types, with the monadic type $\{A\}$ arising as the composite $\uparrow\downarrow P$.

The distinction between positive and negative corresponds to a value/computation distinction (c.f., [24, 44]). While the LNL calculus is also built on an adjunction, this adjunction reflects the linear/nonlinear distinction and not the value/computation distinction. As a result, our language is pure (validating the full $\beta\eta$ theory for all the connectives), and we can freely mix positive and negative types (e.g., $A \otimes B \multimap C$).

However, polarization remains interesting from both purely theoretical and more practical points of view. Spiwack [39] gives *dependent L*, a polarized, linear, dependent type theory from a purely proof-theoretic perspective. The key observation is that polarized calculi have a weaker notion of substitution (namely, variables stand for values, rather than arbitrary terms), and as a result, (a) dependent types may depend only on values (which forbids large eliminations), but (b) since contraction of closed values is "safe" (since no computations can occur in types), linear variables may be freely duplicated when they occur in types.

Similar ideas are also found in the F^* language of Swamy et al. [41]. This is an extension of the $\mathrm{F}^\#$ language with type dependency. Since F^* is an imperative language in the ML tradition, for soundness dependency is restricted to value dependency, just as in dependent L. However, the treatment of dependency on linear variables is somewhat different — F^* has a universe of proof-irrelevant (intuitionistic) types, and affine variables can appear freely in proof-irrelevant types.

Imperative Programming with Capabilities In addition to languages exploring "full-spectrum" type dependency (including Hoare Type Theory, which we discuss in Section 3.3), there are also a number of languages which have looked at using linear types as a programming feature.

ATS with linear dataviews, by Zhu and Xi [45], is not a dependently-typed language, since it very strictly segregates its proof and programming language. However, it has an extremely extensive language of proofs, and this proof language includes linear types, which are used to control memory accesses in a fashion very similar to L3 or separation logic. Interestingly, the entire linear sublanguage is computationally irrelevant – linear proofs are proofs, and hence erased at runtime.

Similarly, the calculus of capabilities of Pottier [37] represents another approach towards integrating linear capabilities with a programming language. As with ATS, capabilities are linear and irrelevant, and this system also includes support for advanced features such as the anti-frame rule. (A simpler proof-irrelevant capability system can be found in the work of Militão et al. [26], who give a kernel linear capability calculus.)

At this point, it should be clear that the combination of proof irrelevance and linearity is something which has appeared frequently in the literature, in many different guises. Unfortunately, it has mostly been introduced on an ad-hoc basis, and is not something which has been studied in its own right. As a result, we cannot give a crisp comparison of our proof irrelevance modality with this other work, since the design space seems very complex.

Other Work One piece of work we wish to draw attention to is the enriched effect calculus of Egger et al. [16]. This calculus is like the (non-dependent) LNL calculus, except that it restricts linear contexts to at most a single variable (so that they are stoups, in Girard's terminology). One striking

coincidence is that EEC has a mixed pair $X\ !\otimes A$, which looks very much like our linear dependent pair $\mathsf{F}x : X.\ A$.

Another direction for future work is to enrich the heap model. Our model of imperative computations is a generalization of Ahmed et al. [2], which in turn builds a logical relation model on top of the basic heap model of separation logic [38]. However, over the past decade, this model of heaps has been vastly extended, both from a semantic perspective [9, 15], and from the angle of verification [14, 22].

References

[1] A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*, pages 57–71. Springer, 2011.

[2] A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

[3] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV*, pages 57–68. ACM, 2007.

[4] A. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program logics for certified compilers*. Cambridge University Press, 2014.

[5] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical computer science*, 70(1):35–64, 1990.

[6] B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computational Structures*, pages 365–379. Springer, 2008.

[7] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic (CSL)*, 1994.

[8] N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *Logic in Computer Science (LICS)*, 1996.

[9] L. Birkedal, K. Støvring, and J. Thamsborg. A relational realizability model for higher-order stateful adts. *The Journal of Logic and Algebraic Programming*, 81(4):491–521, 2012.

[10] I. Cervesato and F. Pfenning. A linear logical framework. *Inf. Comput.*, 179(1):19–75, 2002.

[11] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245, 2011.

[12] R. L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics*, volume 24, pages 21–37. Elsevier, 1985.

[13] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[14] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. *ACM SIGPLAN Notices*, 48(1):287–300, 2013.

[15] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[16] J. Egger, R. E. Møgelberg, and A. Simpson. Enriching an effect calculus with linear types. In *Computer Science Logic*, pages 240–254. Springer, 2009.

[17] M. H. Escardó. Joins in the frame of nuclei. *Applied Categorical Structures*, 11(2):117–124, 2003.

[18] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[19] J.-Y. Girard. Linear logic: Its syntax and semantics. In *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Notes*. CUP, 1995.

[20] R. Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14(1):71–84, 1992.

[21] A. Hobor and J. Villard. The ramifications of sharing in data structures. *ACM SIGPLAN Notices*, 48(1):523–536, 2013.

[22] J. B. Jensen and L. Birkedal. Fictional separation logic. In *Programming Languages and Systems*, pages 377–396. Springer, 2012.

[23] A. Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects*. PhD thesis, 2004.

[24] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[25] P. Martin-Lof and G. Sambin. *Intuitionistic type theory*. Bibliopolis Naples, 1984.

[26] F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *Proceedings of the ACM SIGPLAN 2014 workshop on Programming languages meets program verification*, pages 15–26. ACM, 2014.

[27] A. Miquel. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In *TLCA*, pages 344–359. Springer, 2001.

[28] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *ICFP*, pages 62–73. ACM, 2006.

[29] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.

[30] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 261–274. ACM, 2010.

[31] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *TPHOLs*, pages 189–206, 2007.

[32] A. Nogin. Quotient types: A modular approach. In *Theorem Proving in Higher Order Logics*, pages 263–280. Springer, 2002.

[33] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(02):215–244, 1999.

[34] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative hoare type theory. In *Programming Languages and Systems*, pages 337–352. Springer, 2008.

[35] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS 2001. Proceedings.*, pages 221–230. IEEE, 2001.

[36] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04): 511–540, 2001.

[37] F. Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23 (1):38–144, Jan. 2013.

[38] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[39] A. Spiwack. A dissection of L, 2014. URL `http://assert-false.net/arnaud/papers`.

[40] K. Svendsen, L. Birkedal, and A. Nanevski. Partiality, state and dependent types. In C.-H. L. Ong, editor, *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2011.

[41] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

[42] M. Vákár. Syntax and semantics of linear dependent types, 2014. URL `http://arxiv.org/abs/1405.0033`.

[43] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In *TYPES*, pages 355–377, 2003.

[44] N. Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153 (1-3):66–96, 2008.

[45] D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *PADL*, pages 83–97, January 2005.