Syntax and Semantics of Quantitative Type Theory

Robert Atkey
University of Strathclyde
robert.atkey@strath.ac.uk

Abstract
We present Quantitative Type Theory, a Type Theory that records usage information for each variable in a judgement, based on a previous system by McBride. The usage information is used to give a realizability semantics using a variant of Linear Combinatory Algebras, refining the usual realizability semantics of Type Theory by accurately tracking resource behaviour. We define the semantics in terms of Quantitative Categories with Families, a novel extension of Categories with Families for modelling resource sensitive type theories.

CCS Concepts • Theory of computation → Linear logic: Type theory;

Keywords Type Theory, Linear Logic

ACM Reference Format:

1 Introduction
Dependent Type Theory promises to combine “programming” and “verification” by combining both in a single system. The implementations Agda [34] and Idris [7] advertise themselves as a “dependently typed functional programming language” and a “general purpose pure functional programming language with dependent types”, respectively. Coq [24] is primarily intended as a proof assistant, but also provides a program extraction facility.

However, when trying to actually use Type Theory as a general purpose programming language, type dependency appears to actively encourage inefficient code. This is caused by Type Theory’s use of variables for two purposes: as information to be used in the formation of types, for example, the type Fin(n) of naturals bounded by n depends on the natural number n when type checking, but not at runtime; and as computational information that is manipulated by programs at runtime. An example is illustrated by the type of the cons operation for length indexed vectors of S:

\[
\text{cons} : \left(n : \text{nat}\right) \rightarrow S \rightarrow \text{vec } n S \rightarrow \text{vec } (\text{succ } n) S
\]

A naive implementation of length indexed vectors will store, for every element: a value \(s\); a tail \(v\); and a natural number \(n\) recording the length of \(v\). If a unary representation of natural numbers is used, then this can easily lead to a runtime representation of vectors that consumes memory space quadratic in the length of the list!

Related to the problem of distinguishing between type formation and computational use of data is distinguishing between different kinds of computational use of data. Such information can be used to generate more efficient code, or to ensure that programs only use computational resources in restricted ways (allowing, for example, in place update of memory). Linear Logic [14] initiated a body of research into such systems. Initially, this recorded zero, single, or multiple uses (made explicit by Mogensen [28]), but recent work in coeffects and quantitative types by Petriček et al. [29], Brunel et al.[8], and Ghica and Smith [13] refined this to track usage via semirings. However, extending these systems to dependent Type Theory is not straightforward due to the conflict between type formation and computational uses.

McBride [25] has recently proposed a resolution to this conflict by combining the work on erasability and quantitative types. His insight is to use the 0 of the semiring to represent information that is erased at runtime, but is still available for use in types (i.e., extensionally). McBride presented a syntax and typing rules for the system, as well as an erasure property that exploits the difference between “not used” and “used”, but does not do anything with the finer usage information available. In this paper, we fix and extend McBride’s system, and present semantic interpretations that fully exploit the usage information.

Contributions
1. Section 2 reformulates McBride’s system as Quantitative Type Theory (QTT) to add dependent tensor products and booleans, and to fix a bug in the original system that caused substitution to be inadmissible.
2. Section 3 presents Quantitative Categories with Families, a novel class of categorical models for interpreting QTT.
3. Section 4 presents several concrete realizability models of QTT as instances of QCwFs. These demonstrate that QTT allows resource sensitive interpretation of terms. Read constructively, these interpretations yield an efficient extraction mechanism with precise control over resource usage.

2 Quantitative Type Theory
Combining dependency with linearity is not straightforward. We motivate McBride’s solution and our formulation of it.

2.1 Dependency and Accountancy
In Martin-Löf Type Theory, the term judgement has the form:

\[
x_1 : S_1, x_2 : S_2, \ldots, x_n : S_n \vdash M : T
\]

From Type Theory’s mixed computational/logical point of view, the context \(x_1 : S_1, x_2 : S_2, \ldots, x_n : S_n\) has two uses. It describes the names used for resources which may be used by \(M\) to construct a resource of type \(T\). It also describes the names used to refer to the extensional meanings represented by those resources used to
form the types $S_2, \ldots, S_n$ and $T$. This dual usage is illustrated in the judgement:

$$\pi : \text{Nat} \times P \vdash x : \text{Fin}(n)$$

(1)

where $\text{Nat}$ is the type of natural numbers, and $\text{Fin}(n)$ is the type of numbers less than $n$. The variables $n$ and $x$ play two different roles: $x$ is used as a reference to a resource that is transferred from the input to the output; and $n$ is used to define the type of $x$. The fact that $n$ is not used in the computing being described is not explicitly recorded.

Linear Logic [14] uses presence or absence in a context to explicitly record used resources. A linear typing judgement,

$$x_1 : X_1, \ldots, x_n : X_n \vdash M : T$$

indicates that the term $M$ uses the resources named by the $x_i$ each precisely once.

Linear Logic’s resource accounting via presence conflicts with type dependency. If a variable’s referenced resources are not used computationally, it must not appear in the context and so is not available for use in type formation. Judgement 1 is thus not linear: both because $n$ is not used in the term, and because it is used twice in types.

To resolve this conflict, several authors have used formulations of Linear Logic that distinguish between unrestricted (“intuitionistic”) variables and restricted (“linear”) variables. For simple types, this originates in Barber work [5], where judgements have two contexts separated by a vertical bar:

$$x_1 : S_1, \ldots, x_n : S_n \vdash y_1 : X_1, \ldots, y_n : X_n \vdash M : Y$$

The variables $x_i$ may be used without restriction, zero, one, or many times. The variables $y_j$ must be used exactly once. Thus, the judgement tracks information about the two different kinds of usage. Cervesato and Pfenning [9], and later Krishnaswami et al. [18] and Vákár [35] adapted this to dependent types. Exploiting the fact that variables in the unrestricted context act as they do in normal Type Theory, there is no problem in allowing types to depend on the variables $x_i$. If we wish to treat elements of $\text{Fin}(n)$ as non-duplicable resources, Judgement 1 can be reformulated as:

$$n : \text{Nat} \vdash x : \text{Fin}(n) \vdash_{x} x : \text{Fin}(n)$$

The variables in the unrestricted context now inherit the ambiguous status of variables from standard Type Theory. They can be used purely contemplatively in types and they can be used for computation. The difference is not recorded in the typing judgement. A further restriction is that types cannot depend on linear variables. There is no way that we can contemplate the extensional content of a linear resource for the purposes of typing, for example:

$$n : \text{Nat} \vdash x : \text{Fin}(n) \vdash (x, \text{refl}(x)) : (y : \text{Fin}(n)) \times (x \equiv y)$$

where we are trying to state that the value we return has the same extensional content as the input by pairing the computational representation of the value with proof of equality.

Systems that rely on split contexts still adhere to Linear Logic’s discipline of using presence or absence in a context to track how a variable is used. This restricts what kinds of “use” we can express: either a variable is used or it is not. To lift this latter restriction, systems that annotate variables of the context with information about how they are used have been proposed by Brunel et al. [8], Ghica and Smith [13], and Petriček et al. [29] (the general idea of marking variable bindings rather than their types was originally called “discharged assumptions” by Terui [33]). In all these systems, information about how variables are used is recorded using a semiring. Semirings are a natural structure to use: addition is used to sum up multiple uses of a variable, and multiplication is used to account for nested use. In the notation we will use in this paper, a judgement in these systems looks like:

$$x_1 : p_1 S_1, \ldots, x_n : p_n S_n \vdash M = T$$

(2)

where the $p_1, \ldots, p_n$ are elements of the semiring indicating how the corresponding variable is used. In these systems, the zero of the semiring is used to indicate that a variable is not used at all: $x \not\in S$ is a complicated way of stating that $x$ may as well be absent.

However, when we move to dependent types, 0 usage variables have a useful meaning. McBride [25] reads the usage annotations $p_1$ as indications of computational usage, so a variable with usage 0 indicates that it has no “run-time” presence, but may still be used in the formation of types. The properties of semirings means that 0 usage is ideal for tracking use in types: we always have $0 + p = p$, so combining a computational use with a use in a type retains the original usage; and $0 p = 0$, so nesting an apparently computational use within a type treats the whole usage as noncomputational.

Thus McBride’s system incorporates both erasure (see also Miquel [26] and Mishra-Linger and Sheard [27]) with linearity.

Term typing judgements now have the form:

$$x_1 : p_1 S_1, \ldots, x_n : p_n S_n \vdash M \sigma : T$$

(3)

where the difference from Judgement 2 is that the output is annotated with a usage $\sigma$, where $\sigma$ is restricted to either be the 0 or the 1 of the semiring. This annotation means that we can construct terms that are only to be used in the formation of types. McBride allowed arbitrary usages $\rho$ on the final colon. However, this yields a system that does not admit substitution as we show in Section 2.3.

We now introduce our formulation of McBride’s system, extending it with a data type of booleans, and dependent tensor product types. McBride presented a bidirectional system suited for implementation, and did not distinguish between proper types and terms representing types. We are interested in a semantical investigation of the theory and so switch to a system with an explicit separation of types and representations of types, and with a “declarative” presentation of the rules.

### 2.1.1 Semirings for Usage Accounting

A semiring $R$ is a structure consisting of a carrier set $R$, binary operations $(+) : R \times R \rightarrow R$, $(\cdot) : R \times R \rightarrow R$ and constants $1, 0 \in R$, such that $(R, +, 0)$ is a commutative monoid, $(R, \cdot, 1)$ is a monoid, $(+)\text{ and } ()$ distribute, and $0 p = p 0 = 0$, for all $p \in R$. We also require that the semiring $R$ is positive, meaning that $p + \pi = 0$ implies $p = 0$ and $\pi = 0$, and has the zero-product property: $p \pi = 0$ implies $p = 0$ or $\pi = 0$. These latter two properties are required for the admissibility of substitution.

Suitable semirings include the zero-one-many semiring $[0, 1, \omega]$, where $\omega + \omega = \omega$ and $\omega \cdot \omega = \omega$; the natural numbers with addition and multiplication; and the boolean semiring $[0, 1]$, which we can read as “erased” and “present”.

### 2.1.2 Presyntax

Quantitative Type Theory (QTT) is defined over the syntactic categories of usages, $\rho, \pi$, precontexts $\Gamma$, preterms $S, T, R$, and preterms
The rules that identify the contexts, types, and terms from the precontexts, pretypes, and preterms are described below. The symbol \( \circ \) denotes the empty precontext, which we omit when writing precontexts with more than zero elements. Preterms contain type information that is usually necessary (e.g., the result type \( T \) in a \( \lambda \)-abstraction). This will be used when we interpret the syntax in our categorical models in Section 3.

Precontexts contain usage annotations, \( \rho \), on each of the constituent variables. Scaling, \( \pi \), is defined by:

\[
\pi(\circ) = \circ \\
\pi(\Gamma, x^\rho : S) = \pi(\Gamma), x^{\pi \rho} S
\]

Scaling is shallow in the sense that usage annotations in pretypes \( S \), if any, are not affected. By the semiring laws, context zero-ing, \( 0\Gamma \), sets all the annotations to 0.

Precontext addition \( \Gamma_1 + \Gamma_2 \) is only defined when \( 0\Gamma_1 = 0\Gamma_2 \):

\[
\circ + \circ = \circ \\
(\Gamma_1, x^{\rho_1} S) + (\Gamma_2, x^{\rho_2} S) = (\Gamma_1 + \Gamma_2), x^{\rho_1 + \rho_2} S
\]

These cases are exhaustive by the requirement that \( 0\Gamma_1 = 0\Gamma_2 \).

### 2.1.3 Contexts, Types, Terms

Contexts are identified within the precontexts by the judgement \( \Gamma \vdash \), defined by the following rules:

\[
\begin{align*}
\text{EMP} & : \Gamma \vdash \circ \\
\odot & : \Gamma \vdash \circ \\
\text{EMP} & : \Gamma \vdash \circ \\
\text{EXT} & : \Gamma \vdash \circ
\end{align*}
\]

where the judgement \( \odot \vdash \circ \) indicates that \( \circ \) is well formed as a type in the context \( \Gamma \). We will see the type formation rules for each type former below. The rule EMP builds the empty context. The rule EXT extends a context \( \Gamma \) with a variable \( x \) of type \( S \), with usage annotation \( \rho \). The annotation \( \rho \) indicates building of a context representing environments that provide for \( \rho \)-many uses of \( x \).

As we shall see below, all type formation rules yield judgements where all the usage annotations in \( \Gamma \) are equal to 0. Thus type formation requires no computational resources.

Term judgements in QTT have the form:

\[
\Gamma \vdash M :^\sigma S
\]

where \( \sigma \in \{0, 1\} \). The binary choice of \( \sigma \) effectively splits the theory into two halves. When \( \sigma = 0 \), we are indicating that we are constructing a term with no need for computational content. By Lemma 2.3 (below) when \( \sigma = 0 \) we will necessarily have that all the usage annotations in the context are 0 too. When \( \sigma = 1 \), we are indicating we are constructing computationally relevant data. The two halves are similar to Pfenning’s “\( \# \)” and “\( \# \)” judgements for terms and irrelevant proofs respectively [30].
As for the dependent function type, type formation does not require any intensional resources. The introduction rules:

$$
\begin{align*}
\Gamma \vdash M : \sigma & \Rightarrow S \\
0\Gamma_1 = 0\Gamma_2 \\
\Gamma_2 \vdash N : T[M/x] & \Rightarrow \sigma = 0 \Leftrightarrow (\pi = 0 \lor \sigma = 0) \\
\pi\Gamma_1 + \pi\Gamma_2 + (M, N)_{x:S,T} \Rightarrow (\pi : S \otimes T) \\
\Rightarrow 0\Gamma + & \Rightarrow 0\Gamma + \pi : \sigma \Rightarrow I
\end{align*}
$$

Resource-wise, building a pair is similar to the App rule above. To build an element of the unit type requires no resources. When we are in the extensional portion of the theory ($\sigma = 0$), there are effectively no restrictions, and these are the normal introduction rules for dependent product and unit types.

The elimination rules are interesting because there are two sets, depending on whether we are in the erased ($\sigma = 0$) or present ($\sigma = 1$) part of the theory. Under erasure, we are free to use the normal first and second projection operators:

$$
\begin{align*}
\Gamma \vdash M : (x : \tau) \otimes T & \Rightarrow \Gamma \vdash M : (x : \tau) \otimes T \\
\Gamma \vdash \text{fst}_{x:S,T}(M) : A & \Rightarrow \Gamma \vdash \text{snd}_{x:S,T}(M) : T[\text{fst}(M)/x]
\end{align*}
$$

We also assume the usual $\beta\eta$ equality rules in the erased fragment.

There is no erased eliminator for the I type; in the absence of intensional information, the I type acts like the “unit” type in a normal type theory, and so has no eliminator. We have an $\eta$-rule for I in the $\sigma = 0$ fragment stating that $*$ is the only inhabitant.

When we take into account the resource content of objects, we are more restricted in how we can eliminate objects of tensor product and unit type. Just as for elimination of $\otimes$ in intensional linear types theories, we must use a pattern matching construct to ensure that both parts of the product are used. Likewise, we must explicitly eliminate elements of the unit type:

$$
\begin{align*}
0\Gamma_1, x : (x : \tau) \otimes T & \Rightarrow U & \Gamma_1 \vdash M : (x : \tau) \otimes T \\
\Gamma_2, x : \sigma \Rightarrow S, y : \tau \Rightarrow T & \Rightarrow N : U[(x, y)/z] & 0\Gamma_1 = 0\Gamma_2 \\
\Gamma_1 + \Gamma_2 + \text{let}_{x:S,T}(x, y) = M & \Rightarrow N : U[M/z] \\
0\Gamma_1, x : I & \Rightarrow U & \Gamma_1 \vdash M : I \\
\Gamma_2 + N : U[x/s] & \Rightarrow 0\Gamma_1 = 0\Gamma_2 \\
\Gamma_1 + \Gamma_2 + \text{let}_{I}(x) = M & \Rightarrow N : U[M/x]
\end{align*}
$$

These constructs are subject to the usual $\beta\eta$ and commuting conversion rules [5]. These constructs are permitted in the $\sigma = 0$ and $\sigma = 1$ fragments, but in the $\sigma = 0$ fragment they are shorthand for $\text{fst}$ and $\text{snd}$:

$$
\begin{align*}
0\Gamma_1, x : (x : \tau) \otimes T & \Rightarrow U & \Gamma_1 \vdash M : (x : \tau) \otimes T \\
0\Gamma_1 \vdash M : (x : \tau) \otimes T & \Rightarrow 0\Gamma_1, x : \sigma \Rightarrow S, y : \tau \Rightarrow N : U[(x, y)/z] \\
0\Gamma_1 + \text{let}_{x:S,T}(x, y) = M & \Rightarrow N : U[M/z]
\end{align*}
$$

By combining $\otimes$ and $I$, we can derive an exponential type $!_{\sigma}A$ for QTT as $!_{\sigma}A = (x : \sigma) \otimes I$.

**Boolean Type** The type Bool is an example of a type of data that can be available at runtime.

$$
\begin{align*}
0\Gamma_1 + & \Rightarrow \text{B-True} \\
0\Gamma_1 \vdash \text{Bool} & \Rightarrow 0\Gamma_1 + \text{true}, \sigma : \text{false} \Rightarrow \text{B-True}
\end{align*}
$$

The introduction rules indicate two pieces of information. First, the 0 usage on the context $\Gamma$ indicates that no resources are required to construct the constants true and false. Second, the result usage annotation $\sigma$ is unrestricted, indicating that we are free to choose whether this is a boolean value that is erased or present.

The elimination rule for booleans is:

$$
\begin{align*}
0\Gamma_1, x : \text{Bool} + T & \Rightarrow \Gamma_1 + M : (T[true] \Rightarrow T[false]) \\
\Gamma_1 \vdash M : T[\text{false}] & \Rightarrow \Gamma_1 + N : \text{Bool} \\
0\Gamma_1 + & \Rightarrow 0\Gamma_1 + \text{B-Elim}
\end{align*}
$$

We ensure that the result type $T$ is wellformed, with 0 usage. Then the two branches $M_f$ and $M_f$ for the true and false case respectively must be typed with the same usage annotations $\Gamma_1$. This follows the same style as for the “additive” connectives in Linear Logic. The actual boolean to be eliminated must be constructed from other resources $\Gamma_2$, but this must have the same typing content ($0\Gamma_1 = 0\Gamma_2$).

**Universes** The type of small types, Set, is an example of a type whose elements occupy no resource, but do have a role in forming types. The type formation rule for Set is similar to the one for Bool:

$$
\begin{align*}
0\Gamma_1 + & \Rightarrow \text{Set} \\
0\Gamma_1 + \text{Set} & \Rightarrow 0\Gamma_1 + \text{El(M)}
\end{align*}
$$

The introduction rules for Set are only available in the $\sigma = 0$ fragment; we can never construct a Set value with runtime presence. For instance, the terms representing dependent function types:

$$
\begin{align*}
0\Gamma_1, x : \text{Set} & \Rightarrow 0\Gamma_1, x : \text{El(M)} + N : \text{Set} \\
0\Gamma_1 + (x : \text{Set}) & \Rightarrow N : \text{Set}
\end{align*}
$$

The other introduction rules are similar, but are omitted for space. Elements of Set become proper types via the “decoder” $\text{El}(\text{false})$:

$$
\begin{align*}
0\Gamma_1 + \text{El(M)} & \Rightarrow 0\Gamma_1 + \text{El(M)} \\
0\Gamma_1 + & \Rightarrow 0\Gamma_1 + \text{El(M)}
\end{align*}
$$

This rules indicates that the use of a code for a type in a type does not require any computational information.

Using Set with our other type formers, we can construct additive product and sum types. For example, $S \times T = (b : \text{Bool}) \Rightarrow \text{El(El(M))}_{\text{Set}(S, T, b)}$.

This concludes our presentation of the context, type and term formation rules of QTT. The remaining rules of QTT concern the type and term equality judgements $\Gamma \vdash S \equiv T$ and $\Gamma \vdash M \equiv N : \tau$, but are omitted for space.

### 2.2 Syntactic Properties of QTT

**Usage annotation manipulation** The following shows how usage annotations and the judgements of QTT interact.

**Lemma 2.1.** Usage annotations do not affect context well formedness: $\Gamma_1 +$ and $0\Gamma_1 = 0\Gamma_2$ implies $I_2 +$.

**Lemma 2.2.** The following $0$-ing rules are admissible:

$$
\begin{align*}
\Gamma \vdash M & \Rightarrow S \\
\Gamma \vdash M \equiv N : S & \Rightarrow \Gamma \vdash M \equiv N : S \\
0\Gamma_1 + & \Rightarrow 0\Gamma_1 + \\
\Gamma \vdash M & \Rightarrow S \\
\Gamma \vdash M \equiv N : S & \Rightarrow \Gamma \vdash M \equiv N : S
\end{align*}
$$

These rules state that we can take any term and produce its “resource free” counterpart in the $\sigma = 0$ fragment. Semantically, this will be modelled by the $U$ functor/functions in our QCwFs in Definition 3.2. Note that it is not possible to go the other way: there
Syntax and Semantics of Quantitative Type Theory

is no way to reconstruct the resource usage, and some constructs such as fat and snd only live in the $\sigma = 0$ fragment.

**Lemma 2.3** (Zero needs nothing). If $Γ' ⊢ M : S$, then $0Γ' = Γ$.

Note that the “other direction” of this lemma does not hold. If we have $0Γ' ⊢ M : S$, then it is not necessarily the case that $σ = 0$. For example, the Tm-Boot-True rule produces judgements of arbitrary usage from no resources.

**Weakening** Weakening allows the insertion of variables into contexts, provided they are $0$ usage.

**Lemma 2.4.** Weakening is admissible:

$$
Γ, Γ' \vdash T \quad 0Γ' \vdash U \quad \text{WEAKEN}
$$

$$
Γ, x : U, Γ' \vdash T
$$

where $Γ'$ is “is context”, $S, S \equiv T, M : S$, or $M \equiv N : S$.

**Substitution** Substitution (Lemma 2.5) allows for a variable $x : S$ to be replaced by a term of type $S$, as long as, in the case of the term judgements, we add the resources it requires multiplied by $ρ$ to the resources already present.

**Lemma 2.5.** The following substitution rules are admissible:

$$
Γ, x : S, Γ' \vdash \quad 0Γ' \vdash N : S \quad \text{WEAKEN}
$$

$$
Γ, Γ'[N/x] +
$$

$$
0Γ_1, x : S, 0Γ' \vdash T \quad Γ_2 \vdash N : S \quad 0Γ_1 = 0Γ_2
$$

$$
0Γ_1, 0Γ'[N/x] + T[N/x]
$$

$$
Γ_1, x : S, Γ' \vdash M : T
$$

$$
Γ_2 + N : S \quad 0Γ_1 = 0Γ_2 \quad σ' = 0 \implies ρ = 0
$$

$$
(Γ_1 + ρΓ_2), Γ'[N/x] + M[N/x] : T[N/x]
$$

$$
Γ_1, x : S, Γ' \vdash M \equiv M' : T
$$

$$
Γ_2 + N : S \quad 0Γ_1 = 0Γ_2 \quad σ' = 0 \implies ρ = 0
$$

$$
(Γ_1 + ρΓ_2), Γ'[N/x] + M[N/x] \equiv M'[N/x] : T[N/x]
$$

**2.3 Inadmissibility of Substitution in McBride’s system**

McBride’s original system allowed for term judgements of the form $Γ \vdash M : T$ where $ρ$ is an arbitrary element of the semiring. We now show that such a formulation with the semiring $\{0, 1, ω\}$ yields a system that is not closed under substitution. A proof attempt to show that substitution is admissible fails on App because we need to take the term $Γ \vdash O : T$, which is being substituted in, and split it into some $Γ_1 \vdash O : U$ and $Γ_2 + O : U$ such that $Γ_1 + Γ_2 = Γ$. This is not possible in general. We illustrate the problem with:

$$
f : (x : A) \to A \vdash \lambda x : A. f x : (x : A) \to A
$$

This term coerces some function $f$ from a type that states it uses its argument once to a type where it uses its argument multiple times. This is despite McBride’s system not explicitly allowing subusing.

The following derivation gives a context into which substitution of Judgement 4 fails. Let $D_1$ and $D_2$ stand for uses of the Var rule to derive $y : A \vdash g : (x : A) \to A$ and $y : A \vdash g : (x : A) \to A + y : A$.

If we attempt to substitute $\lambda x : A. f x$ for $g$, then to push the term into the two halves of the top level application, we need to split Judgement 4 according to the equation $ω = 1 + ω$. We already have a version of usage $ω$, but the following judgement is not derivable:

$$
f \vdash (x : A) \to A \vdash \lambda x : A. f x \vdash (x : A) \to A
$$

Indeed, it is not possible to derive $y : A \vdash f : (x : A) \to A$ due to the mismatch between $y$’s $ω$ annotation, and $f$’s 1 requirement. The implicit weakening in Judgement 4 is not accessible without an intervening $λ$-abstraction. This problem is fixed by only allowing 0 or 1 usage on terms, which prohibits this kind of implicit “subsuming”. An alternative solution would be to allow subusing. This is something we plan to investigate.

**3 Quantitative Categories with Families**

Giving a model for dependent Type Theory is a delicate affair due to the complex mutual definition of contexts, types and terms. We build on the existing notion of Categories with Families (CwFs) [11, 15] to define Quantitative Categories with Families (QCwFs) suitable for modelling QTT. Every QCwF contains a CwF, so we first recall their definition.

**Definition 3.1.** A Category with Families (CwF) is a six-tuple $(C, Ty, Tm, \cdot, \cdot, \cdot, \cdot)$, where:

1. $C$ is a category with a chosen terminal object $T$;
2. For $Δ ∈ ObC$, a collection $Ty(Δ)$ of semantic types;
3. For $Δ ∈ ObC$ and $S ∈ Ty(Δ)$, a collection $Tm(Δ, S)$ of semantic terms;
4. For every $f : Δ → Δ'$ in $C$, a function $-f : Ty(Δ') → Ty(Δ)$, and for $S ∈ Ty(Δ')$, a function $-f : Tm(Δ', S) → Tm(Δ, S(f))$, such that both assignments preserve identities and composition.
5. For each object $Δ$ in $C$ and $S$ in $Ty(Δ)$ an object $Δ.S$ (called the comprehension of $S$) such that there is a bijection natural in $Δ'$:

$$
C(Δ', Δ.S) ≅ ([f : M] : f : Δ' → Δ, M ∈ Tm(Δ', S(f))]
$$

Given $f : Δ' → Δ$ and $M ∈ Tm(Δ', S(f))$, we write $f(M)$ for the associated morphism $Δ' → Δ.S$ in $C$. Conversely, given a morphism $f : Δ' → Δ.S$ in $C$, we write $f^{+1} : Δ' → Δ$ and $f^{+2} ∈ Tm(Δ', S(f^{+1}))$ for the associated morphism and semantic term.

The following definitions derive projection and weakening from Definition 3.1, and also define the semantic counterpart of a substitution of a term for a variable. We will require usage annotated counterparts of these below:

1. For $Δ ∈ ObC$ and $S ∈ Ty(Δ)$, the first projection morphism $p_{Δ:S} : Δ.S → Δ$ is defined as $p_{Δ:S} = id^+_{Δ.S}$.
2. For $Δ ∈ C$ and $S ∈ Ty(Δ)$, the second projection $ν_{Δ,S} ∈ Tm(Δ,S,p_{Δ,S})$ is defined as $ν_{Δ,S} = id^2_{Δ.S}$. 
3. For any $\Delta, \Delta'$ in $C$, $S \in Ty(\Delta)$ and morphism $f : \Delta' \rightarrow \Delta$, the weakening of $f$ by $S$, $wk(f, S) : \Delta'.S(f) \rightarrow \Delta.S$ is defined as $wk(f, S) = (f \circ pr_{\Delta'.S(f)},\nu_{\Delta'.S(f)})$.

4. For any $\Delta$ in $C$, $S \in Ty(\Delta)$ and $M \in Tm(\Delta, S)$, we define the morphism $M : \Delta \rightarrow \Delta.S$ as $M = (id_{\Delta}, M)$.

The definition of a Quantitative Category with Families incorporates a CWF, and layers over it facilities for representing contexts, simultaneous substitutions and terms, all with usage information.

**Definition 3.2.** For a usage semiring $R$, an $R$-Quantitative Category with Families ($R$-QCWF) consists of the data:

1. A CWF (Definition 3.1) $(C, Ty, Tm, 1, \cdot, \cdot, (-,) )$.
2. A category $L$, whose objects will be used to interpret contexts with resource annotations and whose morphisms will interpret simultaneous substitutions, together with a faithful functor $U : L \rightarrow C$.
3. (Addition structure) Let $L \times L \rightarrow C$ be the pullback of $f \rightarrow C \leftarrow C$. We require a functor $(+) : L \times L \rightarrow L$ such that $U(\Gamma_1 + \Gamma_2) = U\Gamma_1 (= U\Gamma_2)$.
4. (Scaling structure) For $\rho \in R$, there is a functor $\rho(-) : L \rightarrow L$ such that $U(\rho(-)) = (\rho(-))$.
5. (Resourced Terms) For $\Gamma$ in $L$ and $S \in Ty(UT)$, there is a collection of semantic resourced terms $RTm(\Gamma, S)$, with injective functions $Ut_{\Gamma, S} : RTm(\Gamma, S) \rightarrow Tm(\Gamma, UT)$. For morphisms $f : \Gamma' \rightarrow \Gamma$, and types $S \in Ty(UT)$, there is a function $- f : RTm(\Gamma, S) \rightarrow RTm(\Gamma', S(f))$. If $U(M(f)) = (U\Gamma(M(f)))$.
6. (Resourced context extension) For $\Gamma$ in $L$, $\rho \in R$ and $S \in Ty(UT)$, there is an object $\Gamma, \rho S$ in $L$ such that $U(\Gamma, \rho S) = UT.S$ and there exist natural transformations:
   - $\text{emp}_{\pi} : \circ \rightarrow \pi \circ$
   - $\text{emp}_{\pi^+} : \circ \rightarrow \pi \circ$
   - $\text{ext}_{\pi} : \pi.\xi(\pi)S \rightarrow \pi.\xi(\Gamma, \rho S)$
   - $\text{ext}_{\pi^+} : (\Gamma_1 + \Gamma_2).\rho(p_1 + p_2)S \rightarrow \Gamma_1.\rho S + \Gamma_2.\rho S$
   - $U(\text{emp}_{\pi}) = U(\text{emp}_{\pi^+}) = \text{id} : \Gamma \rightarrow \Gamma, U(\text{ext}_{\pi}) = \text{id}, U(\text{ext}_{\pi^+}) = \text{id}.$
7. Resourced counterparts of the projection, weakening, and term substructure of the underlying CWF:
   - $\text{a.} \Gamma, S \in Ty(UT)$, there is a morphism $pr_{\Gamma, S} : \Gamma.0S \rightarrow \Gamma$ such that $U(pr_{\Gamma, S}) = p_{UT.0S}$.
   - $\text{b.} \Gamma, S \in Ty(UT)$, there is an element $v_{\Gamma, S} \in RTm(\Gamma.0S, S(p_{UT.0S}))$ such that $U(v_{\Gamma, S}) = v_{UT.0S}$.
   - $\text{c.} \Gamma, S \in Ty(UT)$, there is a morphism $wk(f, \rho S) : \Gamma.\xi(\Gamma, \rho S UC(f)) \rightarrow \Gamma.\rho S$ such that $U(wk(f, \rho S)) = wk(U(f), S)$.
   - $\text{d.} \Gamma, S \in Ty(UT)$, there is a morphism $\pi M : \Gamma.\rho p_1 + p_2 \rightarrow \Gamma.\rho S$ such that $U(\pi M) = U\Gamma.M$.

Note that $7d$ does not imply $7e$ in this definition. As noted after Lemma 2.2, it is not possible to go from unresourced terms to resourced terms.

The following proposition highlights the fact that the usage information present in QTT/QCWFs is a refinement of the usual structure of Type Theory. It is possible to completely ignore this refinement and interpret the system in any model of Type Theory.
Interpretation of QTT in a QCwF We follow Hofmann [15]'s presentation of the interpretation of Type Theory in CwF by first defining a partial interpretation of the presyntax, and then showing that this interpretation is well defined on well typed presyntax. For contexts with all annotations 0, types, and terms in the $\sigma = 0$ fragment, the interpretation is as it is for unannotated Type Theory. This fragment is self contained by Lemma 2.3.

To interpret contexts with resource annotations, and terms in the $\sigma = 1$ fragment, we use the additional structure. We interpret precontexts with resource annotations, $\Gamma$ as objects $[[\Gamma]]$ of $L$ and simultaneously partially define two families of morphisms for splitting and distributing resource annotations, using the emp and ext morphisms of the QCwF.

$$s_{\Gamma_1, \Gamma_2} : [[\Gamma_1 + \Gamma_2]] \rightarrow [[\Gamma_1]] + [[\Gamma_2]]$$

$$d_{\sigma, \Gamma} : [[\pi \Gamma]] \rightarrow \pi [[\Gamma]]$$

These morphisms "rearrange" the interpretations of contexts to allow the use of the categorical combinators that interpret terms. In the realisability semantics in Section 4, they will be directly interpreted as rearrangement of resources.

Preterms in the $\sigma = 1$ fragment are partially assigned interpretations with respect to some type assignment $\Delta$ (i.e. a precontext without usage annotations). The interpretation $[[\Delta; M]]$ yields a pair $(\Gamma, t)$ of a precontext with usage annotations matching the variables and types in $\Delta$, and a resourceful semantic term $t$. We write $[[\Gamma,t]; M]$ for the partial assignment of resourceful semantic terms to preterms such that the usage annotations returned by the interpretation match the precontext $\Gamma$. The actual interpretation of preterms is defined by induction over their structure, using $\nu_{\tau, s}$ and $\omega_{f, pS}$ to interpret variables and the appropriate structure for each type's introduction and elimination rules.

Soundness of the interpretation is stated as the following theorem, similar to [15]. The challenge in proving this theorem is in showing that weakening and substitution are soundly interpreted in the $\sigma = 1$ fragment. This is accomplished by showing that their interpretations are tracked by the interpretations of weakening and substitution in the underlying CwF.

Theorem 3.7. 1. If $\Gamma \vdash \tau$, then $[[\Gamma]]$ is an object of $L$;
2. If $\Gamma \vdash S$, then $[[\Gamma; S]] = [[\Gamma]]$;
3. If $\Gamma \vdash S \equiv T$, then $[[\Gamma; S]] = [[\Gamma; T]]$;
4. If $\Gamma \vdash M \equiv S$, then $[[\Gamma; M]] = [[\Gamma; S]]$;
5. If $\Gamma \vdash M \equiv N \uparrow S$, then $[[\Gamma; M]] = [[\Gamma; N]]$.

Comparing R-QCwFs to other models Vákár [35] gives a categorical semantics for linear dependent Type Theory with split contexts as we described in Section 2.1. His models consist of indexed symmetric monoidal closed categories $H : \text{C}_{\text{op}} \rightarrow \text{SMCCat}$ with a comprehension structure. In contrast to QCwFs, Vákár's models permit more freedom in choosing how to interpret linear types and linearity; there is no requirement that every construct in the linear portion erases to a non-linear representative, as QCwFs require via the $U$ functor. QCwFs are intended to model a situation like QTT where we wish to remain with normal Type Theory for the purposes of reasoning at the type level, and allow unrestricted appearance of terms in types, but we also wish to be able to give resource-aware realisations of programs. We substantiate this claim in the next section with realisability models of QTT. We suspect that QCwFs are more specific in that every QCwF (augmented with a $\otimes$-product type former) forms one of Vákár's indexed symmetric monoidal categories.

4 Realisability Models of QTT

In our informal descriptions of QTT and QCwFs, we have distinguished between the erased and present, or resourced and unre-source fragments of the system. We make this distinction formal by constructing realisability models of QTT. Resource sensitive computational content will be represented in the model by realisers from a R-Linear Combinatory Algebras, a refinement of Abramsky, Haghverdi and Scott's Linear Combinatory Algebras [3].

4.1 R-Linear Combinatory Algebras R-Linear Combinatory Algebras ($R$-LCAs) are extensions of $BCI$-algebras.

Definition 4.1. A $BCI$-algebra $(A, ( ), B, C, I)$ consists of sets $A$, a binary operation $( )$ called “application”, and written left associative, and $B, C, I \in A$, such that $B \cdot x \cdot y \cdot z = x \cdot (y \cdot z)$, $C \cdot x \cdot y \cdot z = x \cdot z \cdot y$, and $I \cdot x = x$.

$BCI$-algebras are the untyped counterpart of a Hilbert style axiomatisation of the $\rightarrow$ fragment of Linear Logic, analogous to the $S$ and $K$ combinators for minimal implicational logic. We think of the elements of $A$ as “computational widgets” that can be connected to one another via application. The $B, C, I$ combinators are a basis for linear implication, $\rightarrow o$, allowing composition ($B$), exchange ($C$), and identity ($I$).

$BCI$-algebras are combinatorially complete: given an expression $M$ with variables and the combinators that contains a variable $x$ exactly once, there is an expression $\lambda x.M$ that does not contain $x$, such that $(\lambda x.M) \cdot N = M[N/x]$. Using this fact, we define an $n$-ary tupling operation:

Definition 4.2. Let $p_1, \ldots, p_n$ be elements of a $BCI$-algebra. Define their $\nu_{\tau, s}$ as $[p_1, \ldots, p_n] = \nu_{\tau, s} \cdot p_1 \ldots p_n$. We use the $\nu_{\tau, s}$ notation for $[x_1, \ldots, x_n] = p \in \nu_{\tau, s}$ for $(\nu_{\tau, s} x_1, \ldots, \nu_{\tau, s} x_n, q)$ to represent pattern matching decomposition of tuples.

Example 4.3 (Graph Models). Let $D$ be a set isomorphic to $\mathbb{N} \times D \times D$ (for example, finite binary trees with natural numbers at the leaves). We elide the injections into $D$ and just write $\nu_{\tau, s}$ for a natural number in $D$ and $(a,b)$ for an element of $D$ made from a pair of elements of $D$. The powerset of $D$, $\mathcal{P}(D)$, can be endowed with the structure of a $BCI$-algebra with booleans. Define application to be:

$$a \cdot b = \{ b \mid a = \beta \}.$$
situation in detail. Let $D$ be as in Example 4.3. The set of partial functions $D \rightarrow D$ can be endowed with the structure of a BCI-algebra. We think of elements of this algebra as memoryless computational devices that input and (possibly) output messages represented as elements of $D$. Define application to be:

$$ p \cdot q = p_0 \cup (p_0 ; q ; \cdots ; q^* ; p_0) $$

where we treat partial functions as subsets of $D \times D$, $p_{ij} = ((a, b) | ((i, a), (j, b)) \in p)$, and we use the standard combinator for relations. Conceptually, we are thinking of $p$ as having two ports, 0 and 1, that are connected to the environment and to $q$, respectively. Messages come in the 0 port and are either responded to $(p_0)$ or are forwarded to $q$ via $p_{01}$, which may result in a dialogue, $(p_{11} ; q^*)$, before being forwarded back to the environment by $p_{10}$. We exploit the fact that we can tag messages in $D$ with 0 or 1 to encode the multiplexing of $p$'s messages. The combinator $I = \{(0, x), (1, x) | x \in D \} \cup \{((1, x), (0, x) | x \in D \}$ it forwards messages from the environment to its argument, and from its argument to the environment. The $B$ and $C$ combinators do similar, if more involved, routing. Abramsky notes [2] that the relations involved for the BCI combinator are always symmetric, yielding a model of reversible computation. However, we will see below in Example 4.12 that the structure required for the boolean type will break this reversibility.

Another Gol situation that is suitable for modelling QTT is the resumptions model presented by Abramsky et al. [5]. In this model, the combinators are no longer necessarily memoryless, and can update their behaviour based on previous inputs. Abramsky has drawn attention to the relationship between this model and concurrency models [1]. Hoshino et al. [17] have used the “memorable” aspect of this situation to further allow for effectful computations. An exciting avenue of future research will be to examine the implications of this model for an effectful QTT.

**Definition 4.5 (R-Linear Combinatory Algebra).** Let $R$ be a usage semiring. An $R$-Linear Combinatory Algebra (RLCA) is a BCI-algebra $(\mathcal{A}, \cdot, B, C, I)$ with a function $!_p : \mathcal{A} \rightarrow \mathcal{A}$, for all $p \in R$ and elements $K, W_{\pi}, D, \delta_{\pi}, F_p, \mathcal{A} \subseteq \mathcal{A}$, such that:

$$ K \cdot x \cdot !_0 y = x $$

$$ W_{\pi} \cdot x \cdot !_{x+p} y = x \cdot !_y y \cdot !_p y $$

$$ \delta_{x \cdot p} \cdot !_p x = !_p x $$

$$ F_p \cdot !_p x \cdot !_p y = !_p (x \cdot y) $$

We also require that, for all $x, y \in \mathcal{A}, !_0 x = !_0 y$.

The condition that $!_0 x = !_0 y$ ensures that we have a canonical representation of erased data in our models. Abramsky et al.’s original LCAs correspond to RLCA for the trivial one-element semiring. Given an LCA, we can build an RLCA for the $R = \{0, 1, \omega\}$ semiring:

**Proposition 4.6.** Let $(\mathcal{A}, \cdot, B, C, I, K, W, D, \delta, F)$ be an LCA. We can construct an $(0, 1, \omega)$-LCA over $(\mathcal{A}, \cdot, B, C, I)$ with $!_{0x} = I, !_1 x = x$ and $!_\omega x = \omega x$.

**Example 4.7.** In the graph model of Example 4.3, let us write $[a_1, \cdots, a_n]$ to stand for a representation of lists as right-nested tuples: $(a_1, \cdots, (a_n, 0, \cdots))$. We (partially) define the appending of two elements of $D$ as:

$$ (a, a') \gg a'' = (a, a' \gg a'') $$

$$ 0 \gg a'' = a'' $$

where $n++ a$ is undefined for $n \neq 0$. We similarly define a list membership predicate $a \in b$. Now we define

$$ !_p = \{[a_1, \cdots, a_n] | a_1, \cdots, a_n \in p\} $$

and

$$ K = \{(a, (\{\}, a)) | a \in A\} $$

$$ W = \{((a_1, (a_2, b)), (a_1 \gg a_2, b)) | a_1, a_2, b \in A\} $$

$$ D = \{(a, a) | a \in A\} $$

$$ \delta = \{(a_1 \gg \cdots \gg a_n, \{a_1, \cdots, a_n\}) | a_1, \cdots, a_n \in A\} $$

$$ F = \{((\{a, (b_1, \cdots, b_n)\}) \mid \forall \forall c : c \in a \land (c, b_i) \in f\} $$

**Example 4.8.** Following Abramsky et al., exponentials in the partial functions Gol model are given by $!p = \{(0, x), (x, y) | (x, y) \in p\}$. Thinking of partial functions $D \rightarrow D$ as having I/O ports, exponentiation multiplexes $\mathbb{N}$-many ports through a single port by tagging messages with a port number. The $K, W, D, \delta, F$ combinators are all defined by manipulating the port numbers to perform the required routing. See Abramsky et al. for details [3].

Another source of $R$-LCAs for the natural number semiring is given by taking a BCI-algebra and representing duplicated resources by tupling:

**Proposition 4.9.** Every BCI-algebra $(\mathcal{A}, \cdot, B, C, I)$ can be given the structure of an $(\mathbb{N}, +, 0, 1)$-LCA, where $!n$ is defined by duplication: $!n x = \{x, \ldots, x\}$, using tupling (Definition 4.2).

We also require structure to model the datatypes of booleans:

**Definition 4.10.** An BCI-algebra $\mathcal{A}_0 \subseteq \mathcal{A}$ has booleans if there are elements $T, F \in \mathcal{A}$ and a function $E : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ such that $E(p, q) \cdot T = p$ and $E(p, q) \cdot F = q$.

**Example 4.11.** In the Graph Model of Example 4.3, booleans are represented by $T = (1)$ and $F = (0)$ and

$$ E(a, b) = \{(1, x) | x \in a\} \cup \{(0, x) | x \in b\} $$

**Example 4.12.** Booleans are represented in the partial functions Gol situation algebra (Example 4.4) by $T(x) = (1, x)$, $F(x) = (0, x)$, and:

$$ E(p, q)(0, x) = (1, x) $$

$$ E(p, q)(1, x) = (0, p(x)) $$

$$ E(p, q)(1, 0) = (0, q(x)) $$

That is, $E(p, q)$ forwards its input to its argument, which tags it with 1 or 0 depending on whether it is true or false. On its return to $E(p, q)$, the appropriate one of $p$ or $q$ is applied before forwarding to the environment. Note that the relations described by $E(p, q)$ are not symmetric, indicating the irreversible nature of following a conditional—there is no general way of working out whether the conditional was true or false given the final output. An interesting avenue for future work is to formulate a version of QTT with a type of reversible booleans, yielding a calculus of dependently typed reversible computation.

### 4.2 R-QCwFs from R-LCAs

We now fix an R-LCA with booleans and describe how to use it to give a realizability model of QTT that makes use of the resource information from the types. Hoshino [16] has already investigated assemblies over BCI-algebras and LCAs in the simply typed setting. Here, we extend this analysis to dependently types.
Interpreting Contexts as Assemblies

For the category $\mathcal{L}$ for interpreting contexts and simultaneous substitutions between them, we use the standard notion of the category of Assemblies (see, e.g., Longley and Normann [22], §3.3.6). An assembly $\Gamma$ is a pair of a set $|\Gamma|$ of extensional meanings, and a binary relation $\vdash \subseteq \mathcal{A} \times |\Gamma|$ such that for all $y \in \Gamma$, there is at least one $a \in \mathcal{A}$ such that $a \vdash y$. The idea is that the relation $a \vdash y$ indicates that $a$ is an implementation of the extensional meaning $y$. A morphism between assemblies $(|\Gamma|, \vdash)$ and $(|\Gamma'|, \vdash')$ is a function $\pi : |\Gamma| \to |\Gamma'|$ that is realisable: there exists an $a' \in \mathcal{A}$ such that for all $y \in |\Gamma|$ and $a_y$, if $a_y \vdash y$, then $a'_y \cdot a_y \vdash \pi(y)$. These definitions yield a category, using the combinator $I$ to realise identities, and $B$ to realise composition. We use the usual category of sets and functions for our CwF $\mathcal{C}$. The functor $U : \mathcal{L} \to \mathcal{C}$ projects out the underlying sets and functions.

Scaling of an assembly $\Gamma = (|\Gamma|, \vdash)$ yields an assembly with $|\pi \Gamma| = |\Gamma|$, and the realisability relation is defined by: $x \vdash_{\pi \Gamma} y$ if $\exists y'. x = z \cdot y'. y \vdash_{\Gamma} y'$. Addition of assemblies $\Gamma_1 = (|\Gamma_1|, \vdash_{\Gamma_1})$ and $\Gamma_2 = (|\Gamma_2|, \vdash_{\Gamma_2})$ (note that they have the same underlying set) is defined as $\Gamma_1 + \Gamma_2 = (|\Gamma_1| + |\Gamma_2|, \vdash_{\Gamma_1 + \Gamma_2})$, where $x \vdash_{\Gamma_1 + \Gamma_2} y$ iff there exist $y_1, y_2$ such that $x = [y_1, y_2]$ and $y = y_1 \cdot y_2$. Scalling and addition of interensional interpretations of contexts does not affect the underlying meaning but does affect the computational resources available. The empty context $\emptyset$ is interpreted using the assembly $\emptyset = (\{\}, \vdash_{\emptyset})$, where $x \vdash_{\emptyset} \emptyset$ if $x = I$. The morphisms $\text{emp}_\pi : \emptyset \to \emptyset + \emptyset + \emptyset$ and $\text{emp}_\pi : \emptyset \to \pi \emptyset$ are the identity as functions, and are realised by elimination of the $I$ combinator by treating it as a zero-element tuple.

Interpreting Types as Families of Assemblies

The formation of types has no computational content, hence no realisers, but types do describe how their own extensional inhabitants are realised. Thus we interpret types as families of assemblies indexed over a set, not an assembly. Given a set $\Delta$, the collection $\text{Ty}(\Delta)$ of semantic types consists of $\Delta$-indexed collections of assemblies: $\{(\Delta, \vdash) : \Delta \Delta \}$. For a set $\Delta$ and a family of assemblies $S \in \text{Ty}(\Delta)$, the semantic terms $\text{Tm}(\Delta, S)$ comprise functions of type $\forall \delta \in \Delta. S(\delta)$, ignoring the realisability information in $S$. The rest of the CwF structure on $\mathcal{C}$ is now standard [15].

Unresourced Semantic Terms

For a set $\Delta$ and a family of assemblies $S \in \text{Ty}(\Delta)$, the semantic terms $\text{TM}(\Delta, S)$ comprise functions of type $\forall \delta \in \Delta. S(\delta)$, ignoring the realisability information in $S$. The rest of the CwF structure on $\mathcal{C}$ is now standard [15].

Resourced Context Extension

Given an assembly $(|\Gamma|, \vdash_{\Gamma})$, a family of assemblies $S \in \text{Ty}(|\Gamma|)$ and a usage $\rho \in R$, the set-theoretic part of context extension is defined as $|\Gamma, \rho S| = \{(y, s) \mid y \in |\Gamma|, s \in S(y)\}$. The realisability relation $\vdash_{\Gamma, \rho S}$ is:

$$x \vdash_{\Gamma, \rho S} (y, s) \iff \exists y', z. x = [y', p z] \wedge y \vdash_{\Gamma} y \wedge z \vdash_{S(y)} s$$

Note the use of $\vdash_{\rho}$ to ensure that there are enough computational resources provided to satisfy the promise made. The natural transformations $\text{ext}_\pi : \pi \Gamma, \rho S \to \pi (\Gamma, \rho S)$ and $\text{ext}_\pi : \pi \Gamma, \rho S \to \pi (\Gamma, \rho S)$ are both the identity as functions, and are realised by rearrangement of tupling.

Resourced Semantic Terms

For an assembly $\Gamma$ and a family of assemblies $S \in \text{Ty}(U\Gamma)$, the resource semantic terms from $\Gamma$ to $S$ consist of functions that are tracked by some realiser in $\mathcal{A}$:

$$\text{RTm}(\Gamma, S) = \{f : \forall y \in |\Gamma|, S(y) \mid \exists x. y \in |\Gamma|, y \vdash_{\Gamma} x \cdot y \vdash_{S(y)} f(y)\}$$

Part (7) of Definition 3.2 is now fulfilled by giving $R$-LCA “programs” that realise the necessary resource rearrangement for each combinator. For example, variables $v_{1, s} \in \text{RTm}(of_1, S(p_{UT, S}))$ are realised by using $K$ to discard the context, and $D$ to “derelict” away the $I$ application.

Dependent Function Types

Finally, we must give realisability interpretations for the type formers. For space, we only treat the case of function types. For a set $\Delta$, the underlying sets of the function space family of assemblies consists of the realisables functions:

$$\forall \pi \Delta S(\delta) = \{f : \forall s \in S(\delta). |\Gamma(\delta, s)| \mid \exists x, s. y \in S(\delta) s \Rightarrow x \cdot I, s \vdash_{\Gamma(\delta, s)} f(s)\}$$

Note the use of $\pi_s$ to model the fact that the function’s realiser is given $\pi$-many copies of the input. The bijection $\Lambda$ is realised using currying and uncurrying in the $\text{BCI}$-algebra. See Hoshino [16].

For the remaining type formers: booleans are realised using the structure we stipulated in Definition 4.10. Dependent tensor product types are realised using tuples and $I_s$. The universe of small types is interpreted as the (large) set of families of small assemblies.

5 Related Work

In our introductory section we have already described the main predecessors of this work. In the split context tradition, where intuitionistic and linear types are kept separate, the original work is Cervesato and Flannen’s [9] Linear Logical Framework, which Vákár [35] provided a categorical semantics for; Krishnaswami et al. [18] also investigated a variant of the split context system, also based on Benton’s Linear Non-Linear Logic [6].

Another approach to linear dependent type theory has recently been proposed by Luo [23], where linear and intuitionistic variables are mixed within the same context, and specialised context merging operations are used to distinguish between extensional and intensional uses. We have not yet closely examined the relationship between their system and QTT.

QTT is a kind of modal dependent type theory, particularly with the graded exponential modality we derived from our tensor product type. Syntax and semantics for modal dependent type theories have been studied by de Paiva and Ritter [10]. Licata, Shulman and Riley [21] have studied a general form of substructural and modal simple types with an eye towards extending them to dependent types. Modal type theories are particularly interesting in the context of Homotopy Type Theory, where they can be used to represent the presence or absence of topological structure on types, as shown by Shulman [32].

McBride’s system [25], of which Quantitative Type Theory is a variant, crucially uses usage annotations on types. Such annotations have already appeared in the work of Petricke et al. [29] and Brunel et al. [8] on coeffects, where they are used to track information about the context in which a computation occurs. It would be interesting to compare the concrete models they give (in particular, Brunel et al. use a realisability model) to ours. Ghica and Smith [13] particularly emphasises the quantitative nature of the annotations, and presents an application to timing information.
We have used Abramsky et al.'s [3] notion of Linear Combinatory Algebra as our notion of intensional computational information to be associated with terms. Their Geometry of Interaction situations provide a large supply of LCAs, however not all of them have the necessary structure to interpret the kind of boolean type that we have included in our calculus, as we saw in Example 4.4. This appears to be related to the problem of interpreting the additive connectives of Linear Logic in their axiomatic approach. Nevertheless, the connection with Geometry of Interaction style models is intriguing, and may help develop the theory of QTT into areas such as reversible computation, as described by Abramsky [2], hardware synthesis, as described by Ghica [12], and effectful computation, as described by Hoshino et al. [17].

6 Conclusions and Future Work
Quantitative Type Theory (QTT) reformulates McBride’s combination of linear and dependent types to fix the inadmissibility of substitution. We defined a class of categorical models, Quantitative Categories with Families, and given instances that interpret the precise intensional usage information non trivially, using R-Linear Combinatory Algebras. In addition to reversible and effectful variants arising from the GoI models, we plan future on applying QTT to implicit computational complexity in the style of Dal Lago and Hofmann [19, 20] and to low-level imperative computation in the style of Ahmed et al. [4].

Acknowledgments
Thanks to Conor McBride, Edwin Brady and James Wood for discussions relating to this paper. This work is dedicated to Orwell the dog. Orwell was a good dog and knew well the difference between zero, one, and many.

References