Counting on Quantitative Type Theory

ANONYMOUS AUTHOR(S)

Quantitative Type Theory provides a mechanism to track and reason about resource usage in dependent type systems. In this paper, we develop a novel version of such a type system, including dependent types, tensor products, additive sums, and a graded modality. Since ordinary operational semantics is resource-agnostic; we develop a heap-based operational semantics and prove a soundness theorem that shows correct usage of resources. Several useful properties, including the ordinary type soundness theorem and single pointer property for linear resources, can be derived from this theorem. We expect that our work will provide a foundation for integrating linearity, irrelevance and dependent types in practical programming languages like Haskell.

ACM Reference Format:
Anonymous Author(s). 2020. Counting on Quantitative Type Theory. 1, 1 (July 2020), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Consider this typing judgment.

$$x{:}^{1}\mathbf{Bool},\ y{:}^{1}\mathbf{Int},\ z{:}^{0}\mathbf{Bool} \vdash \mathbf{if}\ x\ \mathbf{then}\ y+1\ \mathbf{else}\ y-1\ :\ \mathbf{Int}$$

Here, the numbers in the context indicates that the variable x is used once in the expression, the variable y is also used only once (although it appears twice), and the variable z is never used at all.

This sentence is a judgment of *Quantitative Type Theory* [Atkey 2018; McBride 2016a], which is concerned with making sure that the *quantities* annotating each variable in the context accurately reflect how they are used at run time. Although this process seems straightforward in this example, when mixed with the features of modern programming languages, this idea turns into a powerful and broadly applicable method of instrumenting type systems with resource and dependency tracking. By abstracting over a domain of resources, this same form of type system can be used for a variety of purposes such as guaranteeing safe memory usage, preventing insecure information flow, quantifying information leakage, and identifying irrelevant computations. Several research languages, such as Idris 2 [Brady 2020] and Agda, are starting to adopt ideas from this domain, and new systems [Gaboardi et al. 2013a; Orchard et al. 2019] are being developed to explore its possibilities.

Our concrete motivation for studying QTT is a desire to merge Haskell's current form of a linear type system [Bernardy et al. 2018] with dependent types [Weirich et al. 2017] in a clean manner. Crucially, the combined system must support *type erasure*: when an argument is quantified irrelevantly, the compiler must be able to eliminate it. Type erasure is key both to support parametric polymorphism and to efficiently execute Haskell programs.

This paper reports on our approach, which synthesizes and refines the ideas of QTT into a novel design. Although our work is strongly inspired by prior and concurrent work, our extensions and

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

analysis provide new insights into this exciting area. More specifically, we develop a dependently-typed programming language integrated with an abstract treatment of resource usage. Our work remains general; although Haskell is our eventual goal, our designs are not specific to that context.

We make the following contributions in this paper:

- As in many systems, our system gains flexibility by abstracting over an algebraic structure that is used to count resources. In Section 2, we describe this structure, a partially-ordered semiring and its properties. This use of a "resource algebra" is standard in this area, although there are subtle differences in its specification that we identify.
- In Section 3 we present a simple type system, with unit, additive sums, multiplicative products and a graded modal type, which uses this algebraic structure to track resource usage. We describe the semantics of this system using a standard, substitution-based small-step operational semantics and show that the system is type sound. This system is again similar to related and concurrent work; our purpose for this section is to establish a foundation to explain our new ideas. However, even at this level, we identify subtleties in the design space.
- Because the standard operational semantics does not include resources, our standard preservation and progress theorems cannot tell us that resource tracking is correct. In Section 4, we design a heap-based operational semantics, inspired by Turner and Wadler [1999], where every variable in the heap has an associated resource from our abstract structure, that models how resources are used during computation. We prove that our type system is sound with respect to this instrumented semantics and track that usage by maintaining a balance equation during computation. This proof tells us that well-typed terms will not get stuck by running out of resources. In the process of showing that this result holds, we identify a key restriction on case analysis that was not forced by the non-resourced version of type safety.
- We find that a quantitative generalization of single pointer property holds for linear resources in Section 5. The single pointer property says that there is only a single pointer to a linear resource. A quantitative generalization of this property states that there is only a single way to refer to a linear resource; meaning, there is only a single pointer to each resource along this way. Such a property would be useful in showing that in-place updates of linear resources are safe.
- We then show how to extend these ideas to dependent types in Section 6. The key idea of this extension is that uses of variables in irrelevant parts of the judgement should not "count". In other words, we use the same type system to check terms and types in this language, but when computing the resource usage of the entire term, we do not include in the usages from types. Our system differs from prior work [Atkey 2018; McBride 2016a] in that it is a more uniform treatment of terms and types. In particular, prior work disables resource checking in types, whereas our system merely ignores resources used in them, the same as any other irrelevant component. We also allow sub-usaging, not considered in these work. One challenge is making sure that our system can express parametric polymorphism, i.e. the quantification over types that are not present at run time.
- We have mechanized the most intricate parts of our proofs. Showing that syntactic properties (substitution, weakening, preservation, progress) of a quantitative dependent type system hold is tedious and error-prone, so we have completed these results with the help of the Coq proof assistant. These proof scripts are available as anonymous supplemental material for this submission.

2 ALGEBRA OF QUANTITIES

The goal of quantitative type theory is to track the demands that computations make on variables that appear in the context. In other words, the type system enables a static accounting of runtime resources "used" in the derivation of terms. This form of type system generalizes linear types (where linear resources must be used exactly once) [Wadler 1990] and bounded linear types (where bounded resources must be used a finite number of times) [Girard et al. 1992], as well as many, many other type systems [Abadi et al. 1999; Abel and Bernardy 2020; Miquel 2001; Orchard et al. 2019; Pfenning 2001; Reed and Pierce 2010; Volpano et al. 1996].

This generality derives from the fact that the type system is parameterized over an abstract algebraic structure of *quantities*, q, to model resources.[1] We want to be able to add and multiply resources and expect that these operations conform to our general understanding of resource arithmetic. An algebraic structure that captures this idea quite nicely is a partially-ordered semiring.

2.1 Partially-ordered semiring

A *semiring* is a set Q with two binary operations, $\_ + \_ : Q \times Q \to Q$ (addition) and $\_ \cdot \_ : Q \times Q \to Q$ (multiplication) and two distinguished elements, 0 (zero) and 1 (one) such that $(Q, +, 0)$ is a commutative monoid and $(Q, \cdot, 1)$ is a monoid; furthermore, multiplication is both left and right distributive over addition and zero is an annihilator for multiplication. Note that a semiring is not a full ring because addition does not have an inverse—we cannot subtract.

We mark the variables in our contexts with quantities to represent demand of resources. In other words, if we have a typing derivation for a term a with free variable x marked with q, we know that a demands q uses of x.

We can weaken the precision of our type system (but increase its flexibility) by allowing the judgement to express a more lenient demand than is actually necessary. For example, we may need to use some variable only once but it may be convenient for the type system to declare that the usage of that variable be unrestricted. To model this notion of *sub-usage*, we need an ordering on the elements of the abstract semiring, reflecting our notion of "leniency". A partial order captures the idea nicely. Since we work with a semiring, such an order should also respect the binary operations of the semiring. Concretely, for a partial order $\leq$ on Q and $q_1, q_2 \in Q$, if $q_1 \leq q_2$, then for any $q \in Q$, we should have $q + q_1 \leq q + q_2$ and $q \cdot q_1 \leq q \cdot q_2$ and $q_1 \cdot q \leq q_2 \cdot q$. A semiring with a partial order satisfying this condition is called a *partially-ordered* semiring.

This abstract structure captures the operations and properties that the type system needs for resource accounting. Because we are working abstractly, we are limited to exactly these assumptions when we reason about the quantities that we use for usage tracking. In trade, it means that we can use a lot more things besides natural numbers to count.

2.2 Examples of partially-ordered semirings

Looking ahead, there are a few semirings that we are interested in. The *trivial semiring* has a single element, and all operations just return that element. Our type system, when specialized to this semiring, degenerates to the usual form of types as the quantities are uninformative.

The *boolean semiring* has two elements, 0 and 1, with the property that $1 + 1 = 1$. A type system can draw quantities from this semiring to distinguish between variables that are used in terms (marked with one) and ones that are unused (marked with zero). Note that in such a type system, the quantity 1 does not correspond to a linear usage: this system cannot count how many times a variable is used, only determine whether a variable has been used.

[1] Because of their abstract nature, quantities are sometimes called modalities, grades, resources, or usages.

There are two different partial orders that make sense for the boolean semiring. If we use the reflexive relation, then this type system precisely tracks relevance. If a variable is marked with 0 in the context, then we know that the variable *must not* be used at runtime, and if it is marked with 1, then we know that it *must* be used. On the other hand, if the partial ordering declares that $0 \leq 1$, then we still can determine that 0 marked variables are unused, but we do not know anything about the usage of 1 marked variables.

The *linearity semiring* has three elements, 0, 1 and ω, where addition and multiplication are defined in the usual way after interpreting ω as "greater than 1". So, we have, $1+1 = \omega$ and $\omega+1 = \omega$ and $\omega \cdot \omega = \omega$. A type system drawing quantities from this semiring can track linear usage by marking linear variables by 1 and unrestricted variables by ω. A suitable ordering for this purpose is the reflexive closure of $\{(0, \omega), (1, \omega)\}$. We don't want $0 \leq 1$ since then we wouldn't be able to guarantee that linear variables in the context are used exactly once. This semiring is the one that makes the most sense for Haskell as it integrates linearity (1 and ω) with irrelevance (0).

The *five-point semiring* has five elements, 0, 1, Aff, Rel and ω, where addition and multiplication are defined in the usual way after interpreting Aff as "1 or less", Rel as "1 or more", and ω as unrestricted. An ordering reflecting this interpretation is the reflexive transitive closure of $\{(0, \text{Aff})), (1, \text{Aff}), (1, \text{Rel}), (\text{Aff}, \omega), (\text{Rel}, \omega)\}$. This semiring can be used to track irrelevant, linear, affine, relevant and unrestricted usage.

Many other examples are possible. Orchard et al. [2019] and Abel and Bernardy [2020] include comprehensive lists of these and other applications including:

- A type system for differential privacy [Reed and Pierce 2010], by using a semiring based on real numbers that represent the information content of computation.
- An information flow type systems, that prevent high-security inputs from affecting the results of low-security computation, by using a semiring of security levels with addition as the join of the lattice, and multiplication as the meet.
- A type system that tracks covariant vs contravariant use of assumptions.

Now, we design a type system over an arbitrary partially-ordered semiring.

3 A SIMPLE QUANTITATIVE TYPE SYSTEM

Our goal is to design a *dependent* usage-aware type system. But, for simplicity, we start with a simply-typed usage-aware system. The grammar and typing rules for this system appear in Figure 1 on page 5. It is parameterized over an arbitrary partially-ordered semiring $(Q, 1, \cdot, 0, +, \leq)$ with quantities $q \in Q$.

The typing judgement for this system has the form $\Delta \,;\, \Gamma \vdash a : A$ and an excerpt of its definition appears at the bottom of the figure. This judgement includes both a standard typing context Δ and a usage context Γ, a copy of the typing context annotated with quantities.

In any typing judgement, we expect the typing context and the usage context to always correspond:

Notation 3.1 (Context dropping). *The notation $\lfloor \Gamma \rfloor$ denotes a context Δ that is like the usage context Γ, but with all quantities dropped. On the other hand $\overline{\Gamma}$ denotes the vector of quantities in Γ.*

Notation 3.2 (Context correspondence). *The notation $\Delta \vdash \Gamma$ denotes that $\Delta = \lfloor \Gamma \rfloor$.*

Lemma 3.3 (Typing context correspondence). *If $\Delta \,;\, \Gamma \vdash a : A$, then $\Delta \vdash \Gamma$.*

This style of including both a plain typing context Δ and its usage counterpart Γ in the judgement is merely for convenience; it allows us to easily tell when two usage contexts differ only in their quantities. There are many alternative ways to express the same information in the type system:

(Grammar)

$$
\begin{array}{llll}
\textit{types} & A, B & ::= & \mathbf{Unit} \mid A \xrightarrow{q} B \mid \Box_q A \\
\textit{terms} & a, b & ::= & x \mid \lambda x{:}^q A.a \mid a\ b \\
 & & \mid & \mathbf{unit} \mid \mathbf{let\,unit}\ = a\ \mathsf{in}\ b \mid \mathbf{box}_q\ a \mid \mathbf{let\,box}\ x = a\ \mathsf{in}\ b \\
 & & \mid & (a, b) \mid \mathbf{let}\ (x, y) = a\ \mathsf{in}\ b \\
 & & \mid & \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 \\
\textit{values} & v & ::= & \mathbf{unit} \mid \lambda x{:}^q A.a \mid \mathbf{box}_q\ a \mid (a, b) \mid \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \\
 & & & \\
\textit{usage contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x{:}^q A \\
\textit{contexts} & \Delta & ::= & \varnothing \mid \Delta, x{:}A
\end{array}
$$

$\boxed{\Delta\ ;\ \Gamma \vdash a : A}$ *(Typing rules for Simple Type System (excerpt))*

$$
\textsc{ST-Var}\ \frac{x \notin \mathrm{dom}\,\Delta \qquad \Delta \vdash \Gamma}{(\Delta, x{:}A)\ ;\ (0{\cdot}\Gamma, x{:}^1 A) \vdash x : A}
\qquad
\textsc{ST-Weak}\ \frac{x \notin \mathrm{dom}\,\Delta \qquad \Delta\ ;\ \Gamma \vdash a : B}{\Delta, x{:}A\ ;\ \Gamma, x{:}^0 A \vdash a : B}
$$

$$
\textsc{ST-Lam}\ \frac{\Delta, x{:}A\ ;\ \Gamma, x{:}^q A \vdash a : B}{\Delta\ ;\ \Gamma \vdash \lambda x{:}^q A.a : (A \xrightarrow{q} B)}
$$

$$
\textsc{ST-App}\ \frac{\Delta\ ;\ \Gamma_1 \vdash a : (A \xrightarrow{q} B) \qquad \Delta\ ;\ \Gamma_2 \vdash b : A}{\Delta\ ;\ \Gamma_1 + q{\cdot}\Gamma_2 \vdash a\ b : B}
\qquad
\textsc{ST-Unit}\ \frac{}{\varnothing\ ;\ \varnothing \vdash \mathbf{unit} : \mathbf{Unit}}
\qquad
\textsc{ST-UnitE}\ \frac{\Delta\ ;\ \Gamma_1 \vdash a : \mathbf{Unit} \qquad \Delta\ ;\ \Gamma_2 \vdash b : B}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash \mathbf{let\,unit}\ = a\ \mathsf{in}\ b : B}
$$

Fig. 1. The simply typed quantitative lambda calculus

we could work with only one usage context Γ and add constraints, or work with a typing context Δ and a separate vector of quantities. Any of these approaches would work.

3.1 Type system basics

We are now ready to start our tour of the typing rules of this system, beginning with rule ST-Var. We see here that a variable x has type A if it has type A in the context—that part is unsurprising. However, as is typical in this style of system the context is is extended to include $0{\cdot}\Gamma$: this notation means that all variables in the context—before x—must have a quantity of 0, which allows weakening to happen at the variable axiom.

Notation 3.4 (Context scaling). *The notation $q{\cdot}\Gamma$ denotes a context Γ' such that, for each $x{:}^r A \in \Gamma$, we have $x{:}^{q\cdot r} A \in \Gamma'$.*

Accordingly, the rule ST-Var states that all variables other than x are not used in the expression x, that is why their quantity is zero. Note also that $x{:}^1 A$ occurs last in the context. If we wish to use a variable that is not the last item in the context, the rule ST-Weak allows us to remove (reading from bottom to top) zero-usage variables at the end of a context.

Any quantitative type system must be careful around expressions that contain multiple sub-expressions. Function application is a prime example, so we examine rule ST-App next. In the rule, we see that the function a has type $A \xrightarrow{q} B$, meaning that it uses its argument, of type A, q times to produce a result of type B. Accordingly, we must make sure that the argument expression b is

able to be used q times. Put another way, we must *multiply* the usage required for b, as recorded in the typing context Γ_2 which is used to check b, by q. We see this in the context used in the rule's conclusion: $\Gamma_1 + q \cdot \Gamma_2$.

This introduces another piece of important notation:

Notation 3.5 (Context addition). *Adding contexts $\Gamma_1 + \Gamma_2$ is defined only when $\lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor$. The result context Γ_3 is such that, for every $x{:}^{q_1} A \in \Gamma_1$ and $x{:}^{q_2} A \in \Gamma_2$, we have $x{:}^{q_1 + q_2} A \in \Gamma_3$. Accordingly, $\lfloor \Gamma_3 \rfloor = \lfloor \Gamma_1 \rfloor$.*

Our approach using two contexts Δ and Γ works nicely here. Because both premises to rule ST-App use the same Δ, we know that the required precondition of context addition is satisfied. The high-level idea here is common in sub-structural type systems: whenever we use multiple sub-expressions within one expression, we must *split* the context. One part of the context checks one sub-expression, and the remainder checks other sub-expression(s).

Example 3.6 (Irrelevant application). *Before considering the rest of the system, it is instructive to step through an example involving a function that does not use its argument, in the context of the linearity semiring. We say that such arguments are* irrelevant. *Suppose that we have a function f, of type $B \xrightarrow{0} A \xrightarrow{1} A$. (Just from this type, we can see that f must be a constant function.) Suppose also that we want to apply this function to some variable x. In this case, define the following contexts*

$$\Delta_0 = f{:}B \xrightarrow{0} A \xrightarrow{1} A \quad \Delta = f{:}B \xrightarrow{0} A \xrightarrow{1} A, x{:}B$$

and usage contexts:

$$\Gamma_0 = f{:}^1 B \xrightarrow{0} A \xrightarrow{1} A \quad \Gamma_1 = f{:}^1 B \xrightarrow{0} A \xrightarrow{1} A, x{:}^0 B \qquad \Gamma_2 = f{:}^0 B \xrightarrow{0} A \xrightarrow{1} A, x{:}^1 B$$

and use them to construct a typing derivation for the application.

$$\textit{ST-App}\ \frac{\textit{ST-Weak}\ \dfrac{\textit{ST-Var}\ \dfrac{}{\Delta_0\ ;\ \Gamma_0 \vdash f : B \xrightarrow{0} A \xrightarrow{1} A}}{\Delta\ ;\ \Gamma_1 \vdash f : B \xrightarrow{0} A \xrightarrow{1} A} \qquad \textit{ST-Var}\ \dfrac{}{\Delta\ ;\ \Gamma_2 \vdash x : B}}{\Delta\ ;\ \Gamma_1 + 0 \cdot \Gamma_2 \vdash f\ x : A \xrightarrow{1} A}$$

Working through the context expression, we see that the computed final context, derived in the conclusion of the application rule is just Γ_1 again. So although the variable x appears free in the expression $f\ x$, it is within an irrelevant part of the term because it is the argument to a constant function. Therefore this use does not contribute to the overall result.

3.2 Data structures

Because this type system subsumes linear types, we must include both an introduction and an elimination form for every type form. This even includes the **Unit** type. We see in the rule that the introduction form, the **unit** value, requires no quantities. This value is eliminated by the pattern matching expression $\mathbf{let\ unit}\ = a\ \mathbf{in}\ b$, a **Unit** typed expression, and adds any quantity used in its computation to those required by b, the next expression to evaluate in sequence.

Graded modal type.

$$\textit{ST-Box}\ \frac{\Delta\ ;\ \Gamma \vdash a : A}{\Delta\ ;\ q \cdot \Gamma \vdash \mathbf{box}_q\ a : \Box_q A} \qquad \textit{ST-LetBox}\ \frac{\Delta\ ;\ \Gamma_1 \vdash a : \Box_q A \quad \Delta, x{:}A\ ;\ \Gamma_2, x{:}^q A \vdash b : B}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash \mathbf{let\ box}\ x = a\ \mathbf{in}\ b : B}$$

The type $\Box_q A$ is called a *usage modal type* (also known as a graded necessity modality[Orchard et al. 2019]). It is introduced by the construct $\mathbf{box}_q\ a$, which uses the expression a q times to build the box. This box can then be passed around as a unit. When unboxed (rule ST-LetBox), the continuation has access to q copies of the contents.

Products.

ST-Pair
$$\frac{\Delta\ ;\ \Gamma_1 \vdash a : A \qquad \Delta\ ;\ \Gamma_2 \vdash b : B}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash (a, b) : A \otimes B}$$

ST-Spread
$$\frac{\Delta\ ;\ \Gamma_1 \vdash a : A_1 \otimes A_2 \qquad \Delta\ ;\ \Gamma_2, x :^1 A_1, y :^1 A_2 \vdash b : B}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = a\ \mathsf{in}\ b : B}$$

The type system includes (multiplicative) products, also known as tensor products. The two components of these pairs do not share variable usage. Therefore the introduction rule adds the two contexts together. These products must be eliminated via pattern matching because both components must eventually be used in the continuation. An elimination form that projects only one component of the tuple would lose the usage constraints from the other component. Note that even though both components of the tuple must be used exactly once, by nesting a modal type within the tuple, programmers can construct data structures with components of varying usage.

Sums.

ST-Inj1
$$\frac{\Delta\ ;\ \Gamma \vdash a : A_1}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_1\ a : A_1 \oplus A_2}$$

ST-Inj2
$$\frac{\Delta\ ;\ \Gamma \vdash a : A_2}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_2\ a : A_1 \oplus A_2}$$

ST-Case
$$\frac{\begin{array}{c} q = q' + 1 \\ \Delta\ ;\ \Gamma_1 \vdash a : A_1 \oplus A_2 \\ \Delta\ ;\ \Gamma_2 \vdash b_1 : A_1 \xrightarrow{q} B \\ \Delta\ ;\ \Gamma_2 \vdash b_2 : A_2 \xrightarrow{q} B \end{array}}{\Delta\ ;\ q{\cdot}\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B}$$

Last, the system includes (additive) sums and case analysis. The introduction rules for the first and second injections are no different from a standard type system. However, in the elimination form, rule ST-Case, the quantities used for the scrutinee must be different than the quantities used (and shared by) the two branches. Furthermore, the case expression may be annotated with a quantity q that indicates how many copies of the scrutinee may be demanded in the branches. Both branches of the case analysis *must* use these subcomponents at least once, as indicated by the $q = q' + 1$ constraint, which forces the scrutinee to be relevant.

3.3 Type soundness

For the language presented above, we define the usual call-by-name reduction relation and excerpt in Figure 2.

With this operational semantics, a syntactic proof of type soundness follows in the usual manner. The substitution lemma needs to compute the resulting context by multiplying the quantities used for the substituted terms by the number of times the variable was needed in the judgement and adding that to the original context of resources, without x.

Lemma 3.7 (Substitution). *If* $\Delta_1\ ;\ \Gamma \vdash a : A$ *and If* $\Delta_1, x : A, \Delta_2\ ;\ \Gamma_1, x :^q A, \Gamma_2 \vdash b : B$ *then* $\Delta_1, \Delta_2\ ;\ \Gamma_1 + q{\cdot}\Gamma, \Gamma_2 \vdash b\{a/x\} : B$.

The preservation theorem shows that all quantities are preserved during computation. The usage context is unchanged with each step.

Theorem 3.8 (Preservation). *If* $\Delta\ ;\ \Gamma \vdash a : A$ *and* $a \rightsquigarrow a'$ *then* $\Delta\ ;\ \Gamma \vdash a' : A$.

$\boxed{a \rightsquigarrow a'}$ *(Small-step reduction)*

$$\textsc{S-AppCong}\ \frac{a \rightsquigarrow a'}{a\ b \rightsquigarrow a'\ b} \qquad \textsc{S-Beta}\ \frac{}{(\lambda x{:}^q A.a)\ b \rightsquigarrow a\{b/x\}} \qquad \textsc{S-UnitCong}\ \frac{a \rightsquigarrow a'}{\mathbf{let\ unit}\ =\ a\ \mathbf{in}\ b \rightsquigarrow \mathbf{let\ unit}\ =\ a'\ \mathbf{in}\ b}$$

$$\textsc{S-UnitBeta}\ \frac{}{\mathbf{let\ unit}\ =\ \mathbf{unit}\ \mathbf{in}\ b \rightsquigarrow b}$$

Fig. 2. Small-step call-by-name reduction (excerpt)

Finally, the progress lemma states that in an empty context, if computation has not finished, then the term is not stuck. (The values are shown in Figure 1).

Theorem 3.9 (Progress). *If* $\varnothing\ ;\ \varnothing \vdash a : A$ *then either* a *is a value or there exists some* a' *such that* $a \rightsquigarrow a'$.

3.4 Discussion and Variations

At this point, the language that we have developed is not too different from that found in prior work, such as Bernardy et al. [2018], Orchard et al. [2019], Wood and Atkey [2020] and Abel and Bernardy [2020]. Most differences are cosmetic especially in the treatment of usage contexts. Of these, the most similar is the concurrently developed Abel and Bernardy [2020], which we compare below.

- First, Abel and Bernardy [2020] includes a slightly more expressive form of pattern matching. Their elimination forms for the box modality and products multiply each scrutinee by some quantity q, providing that many copies of its subcomponents to the continuation, as in our rule ST-Case. For simplicity, we have omitted this feature, it is not difficult to add.
- Second, Abel and Bernardy [2020] includes a way for the judgement to be less precise about the usage required in a computation, called *sub-usaging*. We will add this rule in Section 5.2. In addition, Abel and Bernardy [2020] require that the semiring include least-upper bounds for the partial order of the semiring. This requirement is not necessary for type soundness, but it does make type checking more compositional.
- Finally, in the rule for **case**, Abel and Bernardy [2020] replace the requirement that $q = q' + 1$ with a requirement that $q \leq 1$. These preconditions are not equivalent in an arbitrary partially-ordered semiring. Furthermore, it turns out that neither condition is motivated by the type soundness theorem stated above: the theorem holds without it. Their condition was instead motivated by their parametricity theorems. Our condition is motivated by the heap soundness theorem that we present in the next section.

In particular, the standard type soundness theorem that we showed above is not very informative since it does not show that the quantities are correctly used. But this is as far as we can get with the usage agnostic small-step reduction relation defined above. To show correct usage of quantities, our reduction relation should include an accounting mechanism for usage. It should show the flow of variable usage; how much we have used up, how much we are left with, etc. A heap based semantics turns out to be suitable for this purpose. Heap semantics has been used to model many languages, including resource-aware ones. We follow Launchbury [1993] and Turner and Wadler [1999] in developing a heap semantics for our language.

$\boxed{[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'}$ *(Small-step reduction relation (excerpt))*

SMALL-VAR

$$\frac{}{[H_1, x \overset{(q+1)}{\mapsto} \Gamma \vdash a : A, H_2]\ x \Rightarrow_S [H_1, x \overset{q}{\mapsto} \Gamma \vdash a : A, H_2\,;\,\mathbf{0}^{|H_1|} \diamond 1 \diamond \mathbf{0}^{|H_2|}\,;\,\varnothing]\ a}$$

SMALL-APPL

$$\frac{[H]\ a \Rightarrow_{S \cup \mathrm{fv}\, b} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'}{[H]\ a\ b \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'\ b}$$

SMALL-APP

$$\frac{x \notin \mathrm{dom}\, H \cup \mathrm{fv}\, b \cup S \qquad a' = a\{x/y\}}{[H]\ (\lambda y{:}^{q} A.a)\ b \Rightarrow_S [H, x \overset{q'}{\mapsto} \Gamma \vdash b : A\,;\,\mathbf{0}^{|H|} \diamond 0\,;\,x{:}^{q'} A]\ a'}$$

SMALL-UNITL

$$\frac{[H]\ a \Rightarrow_{S \cup \mathrm{fv}\, b} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'}{[H]\ \mathbf{let\ unit}\ = a\ \mathsf{in}\ b \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ \mathbf{let\ unit}\ = a'\ \mathsf{in}\ b}$$

SMALL-UNITE

$$\frac{}{[H]\ \mathbf{let\ unit}\ = \mathbf{unit}\ \mathsf{in}\ b \Rightarrow_S [H\,;\,\mathbf{0}^{|H|}\,;\,\varnothing]\ b}$$

Fig. 3. Heap semantics

NOTATION 4.1. *The notation $\mathbf{0}^n$ denotes a vector of 0's of length n. When n is clear from the context, we simply write $\mathbf{0}$. The notation $\mathbf{u}_1 \diamond \mathbf{u}_2$ denotes vector concatenation. Here fv a stands for the free variables of a.*

4 HEAP SEMANTICS

A heap semantics shows how a term evaluates when the free variables of the term are assigned to some terms. The assignments are stored in a heap, represented here as an ordered list. We associate an *allowed usage*, basically an abstract quantity of resources, to each assignment. We change these quantities as the evaluation progresses. For example, a typical call-by-name reduction goes like this:

$$\begin{aligned}
& [x \overset{3}{\mapsto} 1, y \overset{1}{\mapsto} x + x](x + y) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{2}{\mapsto} 1, y \overset{1}{\mapsto} x + x]1 + y && \textit{look up value of } y\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{2}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (x + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + 1) && \textit{addition step} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + 2 && \textit{addition step} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]3
\end{aligned}$$

4.1 Reduction Relation

The reduction above is expressed informally as a sequence of pairs of heaps H and expressions a. We formalize this relation using the following judgment, which appears in Fig 3.

$$[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$$

The meaning of this relation is that the term a uses the resources of heap H and steps to a' with H' being the left-over heap. The relation also maintains additional information about evaluation, which we explain below.

Heap assignments are of the form $x \overset{q}{\mapsto} \Gamma \vdash a : A$, associating a variable with its allowed usage q and assignment a. The context Γ and type A help in the proof of our soundness theorem (4.11). For a heap H, we use $\lfloor H \rfloor$ to extract just the list of underlying assignments. For example, for $H = [x \overset{q}{\mapsto} \Gamma \vdash a : A]$, we have $\lfloor H \rfloor = [x \mapsto a]$.

Because we use a call-by-name reduction, we don't evaluate the terms in the heap; we just modify the quantities associated with the assignments as they are retrieved from the heap. Therefore, after any step, H' will contain all the previous assignments of H, possibly with different usages. Furthermore, a beta-reduction step may also add new assignments to H'. To allocate new variable names appropriately, we need a support set S in this relation; fresh names are chosen avoiding the variables in this set. We keep track of these new variables that are added to the heap along with the resources of their assignments using the usage context Γ'.[2] Therefore, after a step $[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, the length of H' is the sum of the lengths of H and Γ'.

Now, because we work with an arbitrary semiring (possibly without subtraction), this heap semantics is non-deterministic. For example, consider a step $[x \overset{q}{\mapsto} a]x \Rightarrow [x \overset{q'}{\mapsto} a]a$, where $q = q' + 1$. Here, we are using x once, so we need to reduce its usage by 1. But in an arbitrary semiring, there may exist multiple new quantities, $q' \neq q''$, such that $q = q' + 1 = q'' + 1$. For example, in the linearity semiring, we have $\omega = 1 + 1 = \omega + 1$. In this case, $[x \overset{\omega}{\mapsto} a]x \Rightarrow [x \overset{1}{\mapsto} a]a$ and $[x \overset{\omega}{\mapsto} a]x \Rightarrow [x \overset{\omega}{\mapsto} a]a$.

The absence of subtraction also means that given an initial heap and a final heap, we really don't know how much resources have been used by the computation. The only way to know this is to keep track of resources while they are being used. The amount of resources used up can be expressed as a quantity vector $\mathbf{u}'$, with the components showing usage at the corresponding variables in H'. (The length of $\mathbf{u}'$ will always be the same as H'.)

The rules in Figure 3 have a one-to-one correspondence with the ordinary small-step rules presented earlier. But there are some crucial differences. The beta rules, instead of immediate substitution, load new assignments into the heap. The var rule changes the usage of the assignment of the corresponding variable. This is the only place where usage gets modified. This is because here alone we use a resource from the heap, the resource being the term the variable is assigned.

The multi-step reduction relation is the transitive closure of the single-step relation.

$$\boxed{[H]\ a \Rightarrow_S^* [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ b} \qquad \textit{(Multi-Step relation)}$$

$$\textsc{Multi-One}\ \frac{[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ b}{[H]\ a \Rightarrow_S^* [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ b} \qquad \textsc{Multi-Many}\ \frac{[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma_1]\ b_1 \quad [H']\ b_1 \Rightarrow_S^* [H''\,;\,\mathbf{u}''\,;\,\Gamma_2]\ b}{[H]\ a \Rightarrow_S^* [H''\,;\,\mathbf{u}' \diamond \mathbf{0}^{|\Gamma_2|} + \mathbf{u}''\,;\,\Gamma_1, \Gamma_2]\ b}$$

[2] Instead of full contexts Γ', we could have just used a list of variable, usage pairs here; but we pass dummy types along with them for ease of presentation later.

Now we look at some properties of this stepping relation. Our heap-semantics is non-deterministic with respect to the usages but it always produces the same final result, assuming that selecting a fresh variable is deterministic.

Lemma 4.2 (Determinism). *If* $[H_1]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$ *and* $[H_2]\ a \Rightarrow_S [H''\,;\,\mathbf{u}''\,;\,\Gamma'']\ a''$ *and* $\lfloor H_1 \rfloor = \lfloor H_2 \rfloor$, *then* $a' = a''$ *and* $\lfloor H' \rfloor = \lfloor H'' \rfloor$.

Let us call heaps H_1 and H_2 similar iff $\lfloor H_1 \rfloor = \lfloor H_2 \rfloor$. Furthermore, for a reduction $[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, call $[\lfloor H \rfloor]a \Rightarrow_S [\lfloor H' \rfloor]a'$ the erased view of reduction. Finally, call a reduction consisting of n steps an n-chain reduction. The above lemma then says that the erased view of every n-chain reduction of any term a, starting with similar heaps, is the same. So if there exists an n-chain reduction of a starting with heap H, we may as well forget all the annotations and evaluate a for n steps starting with $\lfloor H \rfloor$. And by this lemma, such an evaluation in an erased environment is deterministic and hence unique. The reduced term that we get is the same. Along with the soundness theorem, this shall give us a deterministic reduction strategy that is correct.

The reduction relation enforces fair usage of resources, leading to the following theorem.

Theorem 4.3 (Conservation). *If* $[H]\ a \Rightarrow_S^* [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, *then* $\overline{H} \diamond \overline{\Gamma'} = \overline{H'} + \mathbf{u}'$.

Here, $\overline{H}$ represents the available resources and $\overline{\Gamma'}$ represents the newly added resources; whereas $\overline{H'}$ represents the resources left and $\mathbf{u}'$ the resources that were consumed. So the theorem says that the initial resources along with those that are added during evaluation, are equal to the remaining resources plus those that were used up.

4.2 Bisimilarity

Now that we have two reduction relations, we want to know how they compare to each other. For comparison, we need to define some terms. We call a heap acyclic iff the term assigned to a variable does not refer to itself or to any other variables appearing subsequently in the heap. Note that our heap-based reduction relation preserves acyclicity. Now, for an acyclic heap H, we define $a\{H\}$ as the term obtained by substituting in a, in reverse order, the corresponding terms for the variables in the heap. This gives us the following bisimilarity lemma. (Note that $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$.)

Lemma 4.4. *If* H *is an acyclic heap and* $[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, *then* $a\{H\} = a'\{H'\}$ *or* $a\{H\} \rightsquigarrow a'\{H'\}$. *Further, if* $[\varnothing]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, *then* $a \rightsquigarrow a'\{H'\}$.

Lemma 4.5. *If* $a \rightsquigarrow a_1$, *then for any* S *such that* $fv\ a \subseteq S$, *we have* $[\varnothing]\ a \Rightarrow_S [H\,;\,\mathbf{u}\,;\,\Gamma]\ a_2$ *and* $a_2\{H\} = a_1$. *Also, if* $a \rightsquigarrow^* a1$, *then for any* S *such that* $fv\ a \subseteq S$, *we have* $[\varnothing]\ a \Rightarrow_S^* [H\,;\,\mathbf{u}\,;\,\Gamma]\ a_2$ *and* $a_2\{H\} = a_1$.

4.3 Heap compatibility

The key thing to note about this heap-based reduction is that now terms can "get stuck" in two different ways: first, because they are not well-typed and second, because the heap does not contain enough resources. Our claim is that if the resources contained in a heap is judged to be "right" for a term by the type system, then the evaluation of the term in such a heap does not get stuck. This would mean that the type system does a proper accounting of the resource usage of terms. Let us

look at the following evaluation:

$$
\begin{aligned}
& [x \overset{2}{\mapsto} 1, y \overset{1}{\mapsto} x + x](x + y) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{1}{\mapsto} x + x]1 + y && \textit{look up value of } y\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (x + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + x) && \textit{look up value of } x\textit{, stuck!}
\end{aligned}
$$

It gets stuck because the starting heap does not contain enough resources for the evaluation of the term. The term needs to use x thrice; whereas the heap contains only two copies of x. The type system knows this information about resource requirement and can judge whether a heap contains enough resources. We express this idea through the compatibility relation. Given a typing judgement $\Delta\ ;\ \Gamma \vdash a : A$, we use the judgement $H \vdash \Delta; \Gamma$ to assert that H is an appropriate environment for evaluating a.

$\boxed{H \vdash \Delta; \Gamma}$ *(Heap Compatibility)*

$$\textsc{Compat-Empty}\quad \frac{}{\varnothing \vdash \varnothing; \varnothing}$$

$$\textsc{Compat-Cons}\quad \frac{H \vdash \Delta; \Gamma_1 + (q \cdot \Gamma_2) \qquad \Delta; \Gamma_2 \vdash a : A \qquad x \notin \operatorname{dom} H \qquad \Delta \vdash \Gamma_1}{H, x \overset{q}{\mapsto} \Gamma_2 \vdash a : A \vdash \Delta, x{:}A; \Gamma_1, x{:}^q A}$$

Example 4.6. *For*

$$\varnothing\ ;\ \varnothing \vdash x_1 : \mathbf{Int}$$
$$x_1{:}\mathbf{Int}\ ;\ x_1{:}^2\mathbf{Int} \vdash x_1 + x_1 : \mathbf{Int}$$
$$x_1{:}\mathbf{Int}, x_2{:}\mathbf{Int}\ ;\ x_1{:}^1\mathbf{Int}, x_2{:}^1\mathbf{Int} \vdash x_1 + x_2 : \mathbf{Int}$$
$$x_1{:}\mathbf{Int}, x_2{:}\mathbf{Int}, x_3{:}\mathbf{Int}\ ;\ x_1{:}^0\mathbf{Int}, x_2{:}^1\mathbf{Int}, x_3{:}^1\mathbf{Int} \vdash x_3 + x_2 : \mathbf{Int}$$

we have,

$$[x_1 \overset{8}{\mapsto} 1, x_2 \overset{3}{\mapsto} x_1 + x_1, x_3 \overset{1}{\mapsto} x_1 + x_2, x_4 \overset{1}{\mapsto} x_3 + x_2] \vdash x_1 :^1 \mathbf{Int}, x_2 :^1 \mathbf{Int}, x_3 :^0 \mathbf{Int}, x_4 :^1 \mathbf{Int}$$

When a heap is compatible with a term, it is a closing multi-substitution.

Lemma 4.7 (Multi-substitution). *If $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$, then $\varnothing\ ;\ \varnothing \vdash a\{H\} : A$.*

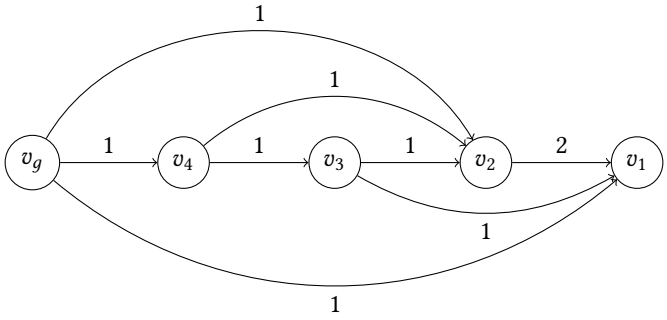
Let us understand the relation $H \vdash \Delta; \Gamma$ better. The heap H is a list of length $|\Gamma|$ with the i^{th} element being $x_i \overset{q_i}{\mapsto} \Gamma_i \vdash a_i : A_i$, where $\Delta_i\ ;\ \Gamma_i \vdash a_i : A_i$ and Δ_i is a prefix-list of Δ of length $i - 1$. This can be seen as a memory layout where x_is correspond to memory locations and a_is to the terms stored in the those locations. The usage q_i then is the number of ways the location x_i can be referenced. We represent this as a memory graph.

4.4 Graphical and algebraic views of the heap

A well-formed heap H where $H \vdash \Delta; \Gamma$ can be viewed as a weighted directed acyclic graph $G_{H,\Gamma}$. Let H contain n assignments with the j^{th} one being $x_j \overset{q_j}{\mapsto} \Gamma_j \vdash a_j : A_j$. Then, $G_{H,\Gamma}$ is a DAG with $(n+1)$ nodes, n nodes corresponding to the n variables in H and one extra node for Γ, referred to as the source node. Let v_j be the node corresponding to x_j and v_g be the source node. Note that Γ_j only contains variables x_1 through x_{j-1}. For $x_i :^{q_{ji}} A_i$ in Γ_j, add an edge with weight $w(v_j, v_i) := q_{ji}$ from

v_j to v_i. We do this for all nodes, including v_g. This gives us a DAG with the topological ordering $v_g, v_n, v_{n-1}, \ldots, v_2, v_1$.

For example 4.6, we have the following memory graph:



For a well-formed heap, we can express the quantities of the assignments in terms of the edge weights of its memory graph. Let us define the length of a path to be the product of the weights along the path. Then, q_j is the sum of the lengths of all paths from v_g to v_j. Note that this is so for the example graph. A path p from v_g to v_j represents a chain of references, with the last one being pointed at v_j. The length of p shows how many times this path is used to reference v_j. The sum of the lengths of all the paths from v_g to v_j then gives a (static) count of the total number of times location v_j is referenced. And this is equal to q_j, the usage of the assignment for v_j in the heap. This means that the usage of an assignment is equal to the (static) count of the number of times the concerned location is referenced. Henceforth, we also refer to q_j as the count of v_j and call the property count balance. Below, we present an algebraic formalization of this property of well-formed heaps.

Notation 4.8. *We use* $\mathbf{0}$ *to denote a vector of* 0*'s of length* n *(when* n *is clear from the context). For a well-formed heap* H *containing* n *assignments of the form* $x_i \overset{q_i}{\mapsto} \Gamma_i \vdash a_i : A_i$*, we write* $\langle H \rangle$ *to denote the* $n \times n$ *matrix whose* i^{th} *row is* $\overline{\Gamma_i} \diamond \mathbf{0}$*. We call* $\langle H \rangle$ *the transformation matrix corresponding to* H*. Matrix operations (over a semiring) are defined in the usual way.*

The transformation matrix for example 4.6 is:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

For a well-formed heap H, the matrix $\langle H \rangle$ is strictly lower triangular. Note that this is also the adjacency matrix of the memory graph, excluding node v_g. The strict lower triangular property of the matrix corresponds to acyclicity of the graph. The count balance property can now be stated.

Lemma 4.9 (Count Balance). *If* $H \vdash \Delta; \Gamma$*, then* $\overline{H} = \overline{H} \times \langle H \rangle + \overline{\Gamma}$.

For example 4.6, we can check that $\overline{H} = \begin{pmatrix} 8 & 3 & 1 & 1 \end{pmatrix}$ satisfies the above equation. Let us understand this equation. For a node v_i in $G_{H,\Gamma}$, we can express the count q_i in terms of the counts of the incoming neighbors and the weights of the corresponding edges. We have, $q_i = \Sigma_j \, q_j w(v_j, v_i) + w(v_g, v_i)$. The right-hand side of this equation represents the static estimate of demand, the amount of resources we shall need while the left-hand side represents the static estimate of supply, the amount of resources we shall have. So $H \vdash \Delta; \Gamma$ is a static guarantee that the heap H shall supply the resource demands of the context Γ.

4.5 Soundness

Now we can state and prove our desired soundness theorem. But to prove this, we need the following lemma.

Lemma 4.10 (Invariance). *If* $H \vdash \Delta; \Gamma_1 + q\cdot\Gamma_2$ *and* $\Delta\ ;\ \Gamma_2 \vdash a : A$ *and* $q = q' + 1$ *for some* q'*, then either* a *is a value or there exists* Γ_3*,* H'*,* $\mathbf{u}'$*,* Γ_0 *and* a' *such that for sufficiently fresh* S*:*

- $[H]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ a'$
- $H' \vdash \Delta, \lfloor\Gamma_0\rfloor; (\Gamma_1, 0\cdot\Gamma_0) + q\cdot\Gamma_3$
- $\Delta, \lfloor\Gamma_0\rfloor\ ;\ \Gamma_3 \vdash a' : A$
- $\overline{\Gamma_3} + \mathbf{u}' + \mathbf{0}\diamond\overline{\Gamma_0} \times \langle H'\rangle = \overline{\Gamma_2}\diamond\mathbf{0} + \mathbf{u}' \times \langle H'\rangle + \mathbf{0}\diamond\overline{\Gamma_0}$

Theorem 4.11 (Soundness). *If* $H \vdash \Delta; \Gamma$ *and* $\Delta\ ;\ \Gamma \vdash a : A$*, then either* a *is a value or there exists* Γ'*,* H'*,* $\mathbf{u}'$*,* Γ_0 *such that for sufficiently fresh* S*:*

- $[H]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ a'$
- $H' \vdash \Delta, \lfloor\Gamma_0\rfloor; \Gamma'$
- $\Delta, \lfloor\Gamma_0\rfloor\ ;\ \Gamma' \vdash a' : A$
- $\overline{\Gamma'} + \mathbf{u}' + \mathbf{0}\diamond\overline{\Gamma_0} \times \langle H'\rangle = \overline{\Gamma}\diamond\mathbf{0} + \mathbf{u}' \times \langle H'\rangle + \mathbf{0}\diamond\overline{\Gamma_0}$

In words, the soundness theorem states that our computations can go forward with the available resources without ever getting stuck. Note that as the term a steps to a', the typing context changes from Γ to Γ'. This is to be expected because during the step, resources from the heap may have been consumed or new resources may have been added. For example, $[x \overset{1}{\mapsto} \mathbf{unit}]x \Rightarrow [x \overset{0}{\mapsto} \mathbf{unit}]\mathbf{unit}$. Though the typing context may change, the new context must be compatible with the new heap. This means that at every step, the dynamics of our language respects our static guarantee on resources. As the computation progresses, the weights in the memory graph change but the count balance property is maintained.

Furthermore, the old context and the new context are related by the equation stated above. Let us look at this equation closely:

$$\begin{aligned} \overline{\Gamma'} \quad &+ \quad \mathbf{u}' \quad &+ \quad \mathbf{0}\diamond\overline{\Gamma_0} \times \langle H'\rangle \\ = \overline{\Gamma}\diamond\mathbf{0} \quad &+ \quad \mathbf{u}' \times \langle H'\rangle \quad &+ \quad \mathbf{0}\diamond\overline{\Gamma_0} \end{aligned}$$

We can understand this equation through the following analogy. The contexts can be seen engaged in a transaction with the heap. The heap pays the context $\mathbf{0}\diamond\overline{\Gamma_0}$ and gets $\mathbf{0}\diamond\overline{\Gamma_0} \times \langle H'\rangle$ resources in return. The context pays the heap $\mathbf{u}'$ and gets $\mathbf{u}' \times \langle H'\rangle$ resources in return. The equation is the "balance sheet" of this transaction.

The ordinary preservation and progress lemma can be derived from this soundness theorem using bisimilarity of the two reduction relations and the multi-substitution property. We sketch the proofs below.

Corollary 4.12. *If* $\varnothing\ ;\ \varnothing \vdash a : A$ *and* $a \rightsquigarrow b$*, then* $\varnothing\ ;\ \varnothing \vdash b : A$*.*

Proof. Since $a \rightsquigarrow b$, for sufficiently fresh S, we have, $[\varnothing]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma']\ b'$ and $b'\{H'\} = b$, by bisimilarity. Since $\varnothing\ ;\ \varnothing \vdash a : A$ and $\varnothing \vdash \varnothing; \varnothing$ and a is not a value, we have $H, \Delta, \Gamma, \Gamma_0, Q, a'$ such that $H \vdash \Delta; \Gamma$ and $[\varnothing]\ a \Rightarrow_S [H\ ;\ \mathbf{u}\ ;\ \Gamma_0]\ a'$ and $\Delta\ ;\ \Gamma \vdash a' : A$, by soundness.

Now, since $[\varnothing]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma']\ b'$ and $[\varnothing]\ a \Rightarrow_S [H\ ;\ \mathbf{u}\ ;\ \Gamma_0]\ a'$, determinism gives us $b'\{H'\} = a'\{H\}$. Since $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a' : A$, by multi-substitution, we have, $\varnothing\ ;\ \varnothing \vdash a'\{H\} : A$. But $a'\{H\} = b'\{H'\}$ and $b'\{H'\} = b$. Therefore, $\varnothing\ ;\ \varnothing \vdash b : A$. □

Corollary 4.13. *If* $\varnothing\ ;\ \varnothing \vdash a : A$*, then* a *is a value or there exists* b*, such that* $a \rightsquigarrow b$*.*

Proof. Since $\varnothing\ ;\ \varnothing \vdash a : A$ and $\varnothing \vdash \varnothing;\varnothing$, we have either a is a value or there exists H, Γ_0, Q, a' such that $[\varnothing]\ a \Rightarrow_S [H\ ;\ \mathbf{u}\ ;\ \Gamma_0]\ a'$. By bisimilarity, $a \rightsquigarrow a'\{H\}$. □

Next we use the soundness theorem to derive some useful properties about usage.

5 APPLICATIONS AND EXTENSIONS

5.1 Applications

Till now, we have developed our theory over an arbitrary partially-ordered semiring. But an arbitrary semiring is too general a structure for deriving theorems we are interested in. For example, the set $\{0, 1\}$ with $1 + 1 = 0$ and all other operations defined in the usual way is also a semiring. But such a semiring does not quite capture our notion of usage since 0 is supposed to mean no usage and 1 (whenever $1 \neq 0$) is supposed to mean some usage. So we restrict our attention to a special kind of semirings. Semirings which satisfy $q_1 + q_2 = 0 \implies q_1 = q_2 = 0$ are called *zerosumfree*. We have the following property for usage tracking with such semirings.

Lemma 5.1. *If* $[H]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ a'$ *and* $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, *then the component* $\mathbf{u}'(x_i) = 0$ *and* $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H'$.

This means that locations with count 0 cannot be referenced during computation. The count for such locations also never changes. Now, if they cannot be referenced, what they contain should not matter. This is true, as we see here:

Lemma 5.2. *If* $[H_1, x_i \overset{0}{\mapsto} \Gamma_i \vdash a : A_i, H_2]\ b \Rightarrow_S [H_1', x_i \overset{0}{\mapsto} \Gamma_i \vdash a : A_i, H_2'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ b'$, *then* $[H_1, x_i \overset{0}{\mapsto} \Gamma_i \vdash a' : A_i, H_2]\ b \Rightarrow_S [H_1', x_i \overset{0}{\mapsto} \Gamma_i \vdash a' : A_i, H_2'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ b'$.

Let us look at locations with count 0 in memory graphs. The length of any path from the source node to such a node must be 0. But all the edge-weights in such a path may be non-zero. This is so because we allow the product of two non-zero elements to be 0. If we disallow this, then there is no path from the source node to such a node (0 weight edges are omitted). If we think in terms of resource usage, it makes sense to place such a restriction. Semirings which satisfy $q_1 \cdot q_2 = 0 \implies q_1 = 0$ or $q_2 = 0$ are called *entire*. Henceforth, we restrict ourselves to entire, zerosumfree semirings.

Lemma 5.3. *If* $H \vdash \Delta;\Gamma$ *and* $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, *then* v_i *belongs to a isolated subgraph (of* $G_{H,\Gamma}$*) that does not contain the source node.*

Along with the soundness theorem, this means that irrelevant resources are never used during computation.

Let us now turn to linearity. For linearity to be meaningful, we need to put more restrictions on our semirings. As an example, consider $1 + 1 = 1$ in the boolean semiring. This goes against our notion of linearity. So we add the following restrictions:

- $q_1 + q_2 = 1 \implies q_1 = 1$ and $q_2 = 0$ or $q_1 = 0$ and $q_2 = 1$
- $q_1 \cdot q_2 = 1 \implies q_1 = q_2 = 1$

We call such semirings linear. For entire, zerosumfree, linear semirings; we have the following property:

Lemma 5.4. *If* $H \vdash \Delta;\Gamma$ *and* $x_i \overset{1}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, *then in* $G_{H,\Gamma}$, *there is a single path* p *from the source node to* v_i *and all the weights on* p *are* 1. *Further, for any node* v_j *on* p, *the subpath is the only path from the source node to* v_j.

Along with the soundness theorem, this gives us a quantitative version of the single pointer property. In words, it means that there is one and only one way to reference a linear resource; any resource along the way has a single pointer to it. This property would enable one to carry out safe in-place update for linear resources.

5.2 Sub-usage

Until now, we have only considered exact usage. But since we work with a partially-ordered semiring, we may allow our contexts to provide more resources than is necessary. Sub-usaging, as it is commonly referred to, allows us to put more resources in our context.

$\boxed{\Delta\ ;\ \Gamma \vdash a : A}$ *(Additional rule)*

$$\textsc{ST-Sub}\ \frac{\Delta\ ;\ \Gamma_1 \vdash a : A \qquad \Gamma_1 \leq \Gamma_2}{\Delta\ ;\ \Gamma_2 \vdash a : A}$$

Notation 5.5 (Context sub-usage). *We write $\Gamma_1 \leq \Gamma_2$ to mean that, for every $x{:}^{q_1}A \in \Gamma_1$, there exists $x{:}^{q_2}A \in \Gamma_2$ with $q_1 \leq q_2$. Furthermore, $\lfloor\Gamma_1\rfloor = \lfloor\Gamma_2\rfloor$. We write $H_1 \leq H_2$ to mean that H_1 and H_2 have the same underlying heap, but with different usage vectors, where $\overline{H_1} \leq \overline{H_2}$.*

Our ordinary stepping relation does not need to change but the heap based reduction relation needs the following rule.

$$\textsc{Small-Sub}\ \frac{[H_1]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma]\ a' \qquad H_1 \leq H_2}{[H_2]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma]\ a'}$$

All the lemma and theorems stated till now hold true, as is, other than than the conservation and the soundness (and invariance) theorems, which need the following modification.

Theorem 5.6 (Conservation). *If $[H]\ a \Rightarrow_S^* [H'\ ;\ \mathbf{u}'\ ;\ \Gamma']\ a'$, then $\overline{H'} + \mathbf{u}' \leq \overline{H} \diamond \overline{\Gamma'}$.*

Theorem 5.7 (Soundness). *If $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$, then either a is a value or there exists Γ', H', $\mathbf{u}'$, Γ_0 such that for sufficiently fresh S:*

- $[H]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ a'$
- $H' \vdash \Delta, \lfloor\Gamma_0\rfloor; \Gamma'$
- $\Delta, \lfloor\Gamma_0\rfloor\ ;\ \Gamma' \vdash a' : A$
- $\overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_0} \times \langle H' \rangle \leq \overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_0}$

This is along expected lines, since with sub-usaging, we don't use all the provided resources. So the exact equality is replaced by less than or equal relation. But if 0 and 1 are minimal elements with respect to this relation, all our previous observations about irrelevant and linear usage hold.

With this, we move on to dependent types.

6 DEPENDENT QUANTITATIVE TYPES

Till now, we have only considered simple types, but in this section we extend to dependent types. First, the syntax: because this section presents a dependently typed version, we combine terms and types into a single syntactic category.

$\boxed{\Delta;\Gamma \vdash a : A}$ *(Typing rules for dependent system (excerpt))*

$$\textsc{T-sub}\ \frac{\Delta;\Gamma_1 \vdash a : A \qquad \Gamma_1 \leq \Gamma_2}{\Delta;\Gamma_2 \vdash a : A}$$

$$\textsc{T-weak}\ \frac{x \notin \mathrm{dom}\,\Delta \qquad \Delta;\Gamma_1 \vdash a : B \qquad \Delta;\Gamma_2 \vdash A : \mathbf{type}}{\Delta, x{:}A;\Gamma_1, x{:}^0 A \vdash a : B}$$

$$\textsc{T-conv}\ \frac{\Delta;\Gamma_1 \vdash a : A \qquad \Delta;\Gamma_2 \vdash B : \mathbf{type} \qquad A \equiv B}{\Delta;\Gamma_1 \vdash a : B}$$

$$\textsc{T-type}\ \frac{}{\varnothing;\varnothing \vdash \mathbf{type} : \mathbf{type}}$$

$$\textsc{T-var}\ \frac{x \notin \mathrm{dom}\,\Delta \qquad \Delta;\Gamma \vdash A : \mathbf{type}}{\Delta, x{:}A;\, 0\cdot\Gamma, x{:}^1 A \vdash x : A}$$

$$\textsc{T-pi}\ \frac{\Delta;\Gamma_1 \vdash A : \mathbf{type} \qquad \Delta, x{:}A;\Gamma_2, x{:}^r A \vdash B : \mathbf{type}}{\Delta;\Gamma_1 + \Gamma_2 \vdash \Pi x{:}^q A.B : \mathbf{type}}$$

$$\textsc{T-lam}\ \frac{\Delta, x{:}A;\Gamma_1, x{:}^q A \vdash a : B \qquad \Delta;\Gamma_2 \vdash A : \mathbf{type}}{\Delta;\Gamma_1 \vdash \lambda x{:}^q A.a : \Pi x{:}^q A.B}$$

$$\textsc{T-app}\ \frac{\Delta;\Gamma_1 \vdash a : \Pi x{:}^q A.B \qquad \Delta;\Gamma_2 \vdash b : A}{\Delta;\Gamma_1 + q\cdot\Gamma_2 \vdash a\ b : B\{b/x\}}$$

$$\textsc{T-Unit}\ \frac{}{\varnothing;\varnothing \vdash \mathbf{Unit} : \mathbf{type}}$$

$$\textsc{T-unit}\ \frac{}{\varnothing;\varnothing \vdash \mathbf{unit} : \mathbf{Unit}}$$

$$\textsc{T-UnitElim}\ \frac{\Delta;\Gamma_1 \vdash a : \mathbf{Unit} \qquad \Delta;\Gamma_2 \vdash b : B\{\mathbf{unit}/y\} \qquad \Delta, y{:}\mathbf{Unit};\Gamma_3, y{:}^r \mathbf{Unit} \vdash B : \mathbf{type}}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathbf{let\ unit}\ =\ a\ \mathbf{in}\ b : B\{a/y\}}$$

Fig. 4. Typing rules for dependent, quantitative type system

$$\begin{array}{rrcl} \textit{terms, types} & a, b, A, B & ::= & \mathbf{type} \\ & & \mid & \Pi x{:}^q A.B \mid x \mid \lambda x{:}^q A.a \mid a\ b \\ & & \mid & \mathbf{Unit} \mid \mathbf{unit} \mid \mathbf{let\ unit}\ =\ a\ \mathbf{in}\ b \\ & & \mid & \Box_q A \mid \mathbf{box}_q\ a \mid \mathbf{let\ box}\ x = a\ \mathbf{in}\ b \\ & & \mid & A \oplus B \mid \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 \\ & & \mid & \Sigma x{:}^q A.B \mid (a, b) \mid \mathbf{let}\ (x, y) = a\ \mathbf{in}\ b \end{array}$$

6.1 Type system

The rules of this type system, shown in Figure 4 on page 17, are inspired by the Pure Type Systems of Barendregt [Barendregt 1993]. However, for simplicity, this system includes only a single sort, **type** and a single axiom **type** : **type**.[3] We annotate Barendregt's system with quantities, as well as add the unit type, sums, sigma types and the box modality. For space, we only omit the rules for sums, Σ-types and the box modality.

The key idea of this design is that quantities only count the *runtime* usage of variables. In a judgement $\Delta;\Gamma \vdash a : A$, the quantities recorded in Γ should be derived only from the parts of a that are needed during computation. All other uses of these variables, whether in the type A, in irrelevant parts of a, or in types that appear later in the context, should not contribute to this count.

[3]This definition corresponds to $\lambda*$, which is 'inconsistent' in the sense that all types are inhabited. However, this inconsistency does not interfere with the syntactic properties of the system that we are interested in as a core for Dependent Haskell.

This distinction is significant because in a dependently-typed system terms may appear in types. As a result, the typing rules must ensure that both terms and types are well-formed during type checking. Therefore, the type system includes premises of the form $\Delta\ ;\ \Gamma \vdash A : \mathbf{type}$, that hold when A is a well-formed type. However, usually such type components do not play a role in computation.

What this means for the type system is that any usage of a context to check an irrelevant component should be multiplied by 0, just like the irrelevant argument in example 3.6. For example, in the rule for variables, any uses of the context Γ to check the type A are discarded (multiplied by 0) in the resulting derivation. Similarly, in the rule for weakening, we check that the type of the weakened variable is well-formed using some context Γ_2 that is compatible with the Γ_1 (same Δ). But Γ_2 doesn't appear in the result of the rule because, for simplicity, we use the simpler Γ_1 instead of the equivalent $\Gamma_1 + 0 \cdot \Gamma_2$. Many rules follow this pattern of checking types with some usage-unconstrained context, including Γ_2 in rule T-conv and rule T-lam, and Γ_3 in rule T-UnitElim.This last rule, implements a form of dependent pattern matching. In this rule, the type of the branch can observe that the that the eliminated term a is equal to the pattern **unit**. To support this refinement, the result type B must type check with a free variable y of an appropriate type.

Note that when we type check types as terms, such as in rule T-pi, we do not use usage-unconstrained contexts. In these cases, we are reasoning about terms that compute types, so we cannot throw away any resources (yet). Instead, we add all resources to the resulting usage context by adding the resources from the two sub-terms together.

Irrelevant quantification. Now consider the quantity r in the rule for checking Π types.

$$\text{T-pi}\quad \frac{\Delta; \Gamma_1 \vdash A : \mathbf{type} \qquad \Delta, x{:}A; \Gamma_2, x{:}^r A \vdash B : \mathbf{type}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \Pi x{:}^q A.B : \mathbf{type}}$$

This quantity records how many times a term of this Π-type uses its argument. We choose to leave this usage unconstrained in the rule: the body of the type can use this argument as many times as it wants! This behavior is sound—the bound variable does not *leak* into the context, so it doesn't reflect any actual resource, just a hypothetical one.

With this rule, we may use 0 to represent parametric polymorphism, by modeling them as irrelevant type arguments. For example, the analogue of the System F type $\forall\alpha.\alpha \to \alpha$, can be expressed in this system as

$$\Pi x{:}^0 \mathbf{type}.x \xrightarrow{1} x$$

This type is only well-formed with the unconstrained rule because, even though the annotation on the variable x is 0, that rule allows x to be used any number of times in the body of the type.

Some versions of irrelevant quantifiers in type theories constrain r to be equal to q, the annotation of the Π-type [Abel and Scherer 2012; Nuyts et al. 2017]. By coupling the usage of variables in body of the abstraction with the result type of the Π, these systems rule out the representation of the type shown above. The benefit for this and other related restrictions is that the systems may support stronger logical reasoning principles about their irrelevant quantifiers (such as internalized parametricity).

In this system, we can model this more restricted form of quantifier with the assistance of the box modality. If, instead of using the type $\Pi x{:}^0 A.B$, we use the type $\Pi x{:}^1 \Box_0 A.B$, we can force the result type to also make no (relevant) use of the argument within B. The box x can be unboxed as many times as desired, but each unboxing must be used exactly 0 times.

It is this distinction between the types $\Pi x{:}^0 A.B$ and $\Pi x{:}^1 \Box_0 A.B$ that motivates our inclusion of both the usage annotation on the Π type itself and the modal type $\Box_q A$. In the simple type system, we can derive usage-annotated functions from linear functions and the box modality: there is no need to annotate any arrow with any quantity other than 1 [Orchard et al. 2019]. However, in the dependent type system, we gain additional expressiveness from including both features. Graded Type Theory (GrTT) takes this to the next level by tracking usages both in types and functions [Moon et al. 2020]. This allows precise control over variable usage while computing in types that can lead to compile time optimizations, encodings of other systems, and controlling parametricity in the style of [Nuyts and Devriese 2018].

6.2 Metatheory

We have proven, in Coq, the following properties about the dependently-typed system.

First, well-formed terms have well-formed types. However, the resources used by such types are unrelated to those of the terms.

Lemma 6.1 (Regularity). *If* $\Delta\ ;\ \Gamma \vdash a : A$ *then there exists some* Γ' *such that* $\Delta\ ;\ \Gamma' \vdash A : \mathbf{type}$.

Next, we generalize the substitution lemma for the simple version to this system, by propagating it through the context and type.

Lemma 6.2 (Substitution). *If* $\Delta_1\ ;\ \Gamma \vdash a : A$ *and* $\Delta_1, x : A, \Delta_2; \Gamma_1, x :^q A, \Gamma_2 \vdash b : B$ *then* $\Delta_1, \Delta_2\{a/x\}; (\Gamma_1 + q{\cdot}\Gamma), \Gamma_2\{a/x\} \vdash b\{a/x\} : B\{a/x\}$.

Furthermore, even though we have an explicit weakening rule in this system, we also show that we can weaken with a zero-annotated fresh variable any where in the judgement, as long as its type is well-formed.

Lemma 6.3 (Weakening). *If* $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash a : A$ *and* $\Delta_2; \Gamma_3 \vdash B : \mathbf{type}$ *then* $\Delta_1, x{:}B, \Delta_2; \Gamma_1, x{:}^0 B, \Gamma_2 \vdash a : A$.

The small-step relation for this language is identical to that of the simply typed version, shown in Figure 2.

Theorem 6.4 (Preservation). *If* $\Delta; \Gamma \vdash a : A$ *and* $a \rightsquigarrow a'$ *then* $\Delta; \Gamma \vdash a' : A$.

Theorem 6.5 (Progress). *If* $\varnothing; \varnothing \vdash a : A$ *then either* a *is a value or there exists some* a' *such that* $a \rightsquigarrow a'$.

Now we develop the heap semantics for the dependent version.

6.3 Heap semantics

The step relation $a \rightsquigarrow a'$ is the same for both simple and dependent types. So can we use the same heap-based reduction for dependent types? Almost. Since we freshen names in the reduction relation, we need to pass the types along. In the simply typed version, this was not required since the types did not depend on terms. But this modification is cosmetic: only to keep the variables aligned. The new relation looks like $[H]\ a{:}\ A \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_0]\ a'{:}\ A'$, where A and A' are dummy types.

With this new relation, we need to extend our determinism lemma.

Lemma 6.6 (Determinism). *If* $[H_1]\ a{:}\ A \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma']\ a'{:}\ A_1$ *and* $[H_2]\ a{:}\ A \Rightarrow_S [H''\ ;\ \mathbf{u}''\ ;\ \Gamma'']\ a''{:}\ A_2$ *and* $\lfloor H_1 \rfloor = \lfloor H_2 \rfloor$, *then* $a' = a''$ *and* $\lfloor H' \rfloor = \lfloor H'' \rfloor$ *and* $A_1 = A_2$.

The bisimilarity remains intact. The multi-substitution lemma changes in the following way:

$\boxed{[H]\ a\colon A \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'\colon A'}$ *(Small-step, excerpt)*

SmallS-Var

$$\frac{}{[H_1, x \overset{(q+1)}{\mapsto} \Gamma \vdash a : A, H_2]\ x\colon A \Rightarrow_S [H_1, x \overset{q}{\mapsto} \Gamma \vdash a : A, H_2\,;\, \mathbf{0}^{|H_1|} \diamond 1 \diamond \mathbf{0}^{|H_2|}\,;\, \varnothing]\ a\colon A}$$

SmallS-AppL

$$\frac{[H]\ a\colon A \Rightarrow_{S \cup \mathrm{fv}\, b} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'\colon B}{[H]\ a\, b\colon A \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'\, b\colon B}$$

SmallS-App

$$\frac{x \notin \mathrm{dom}\, H \cup \mathrm{fv}\, b \cup S}{[H]\ (\lambda y{:}^q A.a)\ b\colon A \Rightarrow_S [H, x \overset{q'}{\mapsto} \Gamma \vdash b : A\,;\, \mathbf{0}^{|H|} \diamond 0\,;\, x{:}^{q'} A]\ a\{x/y\}\colon A\{x/y\}}$$

Fig. 5. Heap semantics

Lemma 6.7 (Multi-substitution). *If $H \vdash \Delta;\Gamma$ and $\Delta\,;\,\Gamma \vdash a : A$, then $\varnothing\,;\,\varnothing \vdash a\{H\} : A\{H\}$.*

We also have a restricted type substitution lemma.

Lemma 6.8 (Type substitution). *If $\Delta\,;\,\Gamma \vdash a : A$, and for sufficiently fresh S, if $[H]\ a\colon A \Rightarrow_S [H_0, H'\,;\,\mathbf{u}'\,;\,\Gamma_0]\ a'\colon A'$ where $\lfloor H\rfloor = \lfloor H_0\rfloor$, we have $A'\{H'\} \equiv A$.*

Here H' denotes the new assignments resulting from the reduction of the term. If we substitute the assigned terms for these new variables in the type, we get equal types.

We now present the soundness theorem for dependent types:

Theorem 6.9 (Soundness). *If $H \vdash \Delta;\Gamma$ and $\Delta\,;\,\Gamma \vdash a : A$, then either a is a value or there exists Γ', H', $\mathbf{u}'$, Γ_0, A' such that for sufficiently fresh S:*

- $[H]\ a\colon A \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma_0]\ a'\colon A'$
- $H' \vdash \Delta, \lfloor\Gamma_0\rfloor;\Gamma'$
- $\Delta, \lfloor\Gamma_0\rfloor\,;\,\Gamma' \vdash a' : A'$
- $\overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_0} \times \langle H'\rangle \leq \overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H'\rangle + \mathbf{0} \diamond \overline{\Gamma_0}$

We can use this soundness theorem to prove preservation and progress. The proof for progress is the same as the one for the simply typed version. We present the proof for preservation below:

Corollary 6.10. *If $\varnothing\,;\,\varnothing \vdash a : A$ and $a \rightsquigarrow b$, then $\varnothing\,;\,\varnothing \vdash b : A$.*

Proof. Since $a \rightsquigarrow b$, for sufficiently fresh S, we have, $[\varnothing]\ a : A \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ b' : B$ and $b'\{H'\} = b$, by bisimilarity. By type substitution, $B\{H'\} \equiv A$.

Since $\varnothing\,;\,\varnothing \vdash a : A$ and $\varnothing \vdash \varnothing;\varnothing$ and a is not a value, we have $H, \Delta, \Gamma, \Gamma_0, Q, a', A'$ such that $H \vdash \Delta;\Gamma$ and $[\varnothing]\ a : A \Rightarrow_S [H\,;\,\mathbf{u}\,;\,\Gamma_0]\ a' : A'$ and $\Delta\,;\,\Gamma \vdash a' : A'$, by soundness. Since $[\varnothing]\ a : A \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ b' : B$ and $[\varnothing]\ a : A \Rightarrow_S [H\,;\,\mathbf{u}\,;\,\Gamma_0]\ a' : A'$, determinism gives us $b'\{H'\} = a'\{H\}$ and $B = A'$.

Since $H \vdash \Delta;\Gamma$ and $\Delta\,;\,\Gamma \vdash a' : A$, by multi-substitution, we have, $\varnothing\,;\,\varnothing \vdash a'\{H\} : A'\{H\}$. But $a'\{H\} = b'\{H'\}$ and $b'\{H'\} = b$. Therefore, $\varnothing\,;\,\varnothing \vdash b : A'\{H\}$. Further, since $B = A'$ and $B\{H'\} \equiv A$, we get $A'\{H\} \equiv A$. Hence, $\varnothing\,;\,\varnothing \vdash b : A$. □

This shows that the ordinary semantics is sound with respect to a resource-aware semantics.

7 DISCUSSION

7.1 Definitional-equivalence and irrelevance

The terms "irrelevance" and "irrelevant quantification" have multiple meanings in the literature. Our primary focus is on erasability, the ability for terms to quantify over arguments that need not be present at runtime. However, this terminology often includes compile-time irrelevance, or the blindness of type equality to such erasable parts of terms. These terms are also related to, but not the same as, "parametricity" or "parametric quantification", which characterizes functions that map equivalent arguments to equivalent results.

One difference between our formulation and a more traditional dependently-typed calculus is that the conversion rule (rule T-CONV) is specified in terms of an abstract equivalence relation on terms, written $A \equiv B$. Our proofs about this system work for any relation that satisfies the following properties.

Definition 7.1. We say that the relation $A \equiv B$ is *sound* if it:

(1) *is equivalence relation*,
(2) *contains the small step relation*, in other words, if $a \rightsquigarrow a'$ then $a \equiv a'$,
(3) *is closed under substitution*, in other words, if $a_1 \equiv a_2$ then $b\{a_1/x\} \equiv b\{a_2/x\}$ and $a_1\{b/x\} \equiv a_2\{b/x\}$,
(4) *is injective for type constructors*, for example, if $\Pi x{:}^{q_1} A_1.B_1 \equiv \Pi x{:}^{q_2} A_2.B_2$ then $q_1 = q_2$ and $A_1 \equiv A_2$ and $B_1 \equiv B_2$ (and similar for $\Box_q A$ and $A \oplus B$),
(5) and *is consistent*, in other words, if $A \equiv B$ and both are values, then they have the same head form.

The standard β-conversion relation, defined as the reflexive, symmetric, transitive and congruent closure of the step relation, is a sound relation.

However, β-conversion is not the only relation that would work. Dependent type systems with irrelevance sometimes erase irrelevant parts of terms before comparing them up to β-equivalence [Barras and Bernardo 2008]. Alternatively, a typed definition of equivalence, might use the total relation when equating irrelevant components [Pfenning 2001]. In future work, we hope to show that any sound definition of equivalence can be coarsened by ignoring irrelevant components in terms during comparison. We conjecture that such a relation would also satisfy the properties above. In particular, our results from Section 5 tell us that such coarsening of the equivalence relation is consistent with evaluation, and therefore contains the step relation.

7.2 Connection to Haskell

The practical, on-the-ground motivation for us in exploring these details of dependent, quantitative type theory is the development of dependent types for GHC/Haskell [Eisenberg 2016; Gundry 2013; Weirich et al. 2019, 2017]. Support for linear types [Bernardy et al. 2018] is included with GHC 8.12. We thus need a way to marry these two features as part of a cohesive whole.

A key ingredient in a successful combination of linear and dependent types for Haskell is to capitalize on the 0 quantity to mean *irrelevant*. Previous work [e.g., Weirich et al. 2017] discusses irrelevant quantification for Dependent Haskell. Irrelevance for Haskell is important for two reasons: it allows us to retain traditional parametric polymorphism even in the face of dependent types, and it allows us to be concrete about type erasure. Haskell, with industrial users who care deeply about performance, must retain its ability to erase types before runtime. By noting where arguments should be quantified irrelevantly, programmers can indicate where they expect type erasure to take

place. This will be a user-facing feature: as Eisenberg [2018] explains,[4] users will explicitly denote whether they want irrelevant quantification or relevant quantification. By marking irrelevant quantification using 0, irrelevance fits in swimmingly with Haskell's current story around linear types.

This current design is essentially an instance of the type system described in this paper, using the linearity semiring. Users can mark arguments with grades 1 or ω, but a grade of 0 is sometimes needed internally. Haskell's kind system supports irrelevance, but not linearity, so the two features do not yet interact. It is only with dependent types that we need to deploy our brand of quantitative types. The current structure will be able to migrate to quantitative type theory with little, if any, backward compatibility trouble for users.

One feature of Haskell's linear types does cause a small wrinkle, though: Haskell supports *multiplicity polymorphism*. An easy example is in the type of `map`, which is `forall m a b. (a #m-> b) -> [a] #m-> [b]`. We see that the function argument to `map` can be either linear or unrestricted, and that this choice affects whether the input list is restricted. We cannot support quantity polymorphism in our type theory, as quantifying over whether or not an argument is relevant would mean that we could no longer compile a quantity-polymorphic function: would the compiled function take the argument in a register or not? The solution is to tweak the meaning of quantity polymorphism slightly: instead of quantifying over *all* possible quantities, we would be polymorphic only over quantities q such that $1 \leq q$. That is, we would quantify over only relevant quantities. This reinterpretation of multiplicity polymorphism avoids the trouble with static compilation. Furthermore, we see no difficulty in extending our quantitative type theory with this kind of quantity polymorphism; in the linear Haskell work, multiplicity polymorphism is nicely straightforward, and we expect the same to be true here, too.

7.3 Abstract Algebraic Generalization

Our type system with graded contexts has operations for addition ($\Gamma_1 + \Gamma_2$) and scalar multiplication ($q\Gamma$) defined over an arbitrary partially-ordered semiring. Furthermore, the partial ordering from the semiring was lifted to contexts $\Gamma_1 \leq \Gamma_2$. However, we can provide reasonable alternative definitions for these operations and relations and all our proofs would still work the same. Here, we layout what constitutes a reasonable definition.

Our contexts are an example of a general algebraic structure, called a partially-ordered left semimodule. We look at this abstract structure and some of its properties. Besides our graded contexts, we shall see that vectors and matrices of quantities also can also be seen through this abstract mathematical lens. This may help in future extensions and applications of the work presented in this paper.

We follow Golan [1999] in our terminology and definitions here.

Definition 7.2 (Left Q-semimodule). Given a semiring $(Q, +, \cdot, 0, 1)$, we say that a left Q-semimodule is a commutative monoid $(M, \oplus, \bar{0})$ along with a left multiplication function $\_ \odot \_ : Q \times M \to M$ such that the following properties hold.

- for $q_1, q_2 \in Q$ and $m \in M$, we have, $(q_1 + q_2) \odot m = q_1 \odot m \oplus q_2 \odot m$
- for $q \in Q$ and $m_1, m_2 \in M$, we have, $q \odot (m_1 \oplus m_2) = q \odot m_1 \oplus q \odot m_2$
- for $q_1, q_2 \in Q$ and $m \in M$, we have, $(q_1 \cdot q_2) \odot m = q_1 \odot (q_2 \odot m)$
- for $m \in M$, we have, $1 \odot m = m$
- for $q \in Q$ and $m \in M$, also, $0 \odot m = q \odot \bar{0} = \bar{0}$.

[4] That proposal was not accepted, as it was deemed premature. Even so, the GHC Steering Committee appreciated the proposal and encouraged it to be re-raised when the time is right.

Graded contexts Γ (with the same $\lfloor\Gamma\rfloor$) satisfy this definition, using the operations as defined before. Another example of a semimodule is Q itself, with $\oplus := +$ and $\odot := \cdot$.

In fact, any Cartesian power of Q is also a left semimodule. Given a semiring Q, let Q^n be the set of n-length vectors of elements of Q. Then Q^n with $\oplus$ and $\odot$ defined componentwise forms a left Q-semimodule.

Next, let us consider the partial ordering of our contexts. The ordering is basically a lifting of the partial ordering in the semiring. But in general, a partial order on a left semimodule needs to satisfy only the following properties.

Definition 7.3 (Partially-ordered left Q-semimodule). Given a partially-ordered semiring $(Q, \leq)$, a left Q-semimodule M is said to be partially-ordered iff there exists a partial order $\leq_M$ on M such that the following properties hold.

- for $m_1, m_2, m \in M$, if $m_1 \leq_M m_2$, then $m \oplus m_1 \leq_M m \oplus m_2$
- for $q \in Q$ and $m_1, m_2 \in M$, if $m_1 \leq_M m_2$, then $q \odot m_1 \leq_M q \odot m_2$
- for $q_1, q_2 \in M$ and $m \in M$, if $q_1 \leq q_2$, then $q_1 \odot m \leq_M q_2 \odot m$.

Note that our ordering of contexts Γ satisfy these properties.

We use matrices on several occasions. Matrices can be seen as homomorphisms between semimodules. Given a semiring Q, an $m \times n$ matrix with elements drawn from Q is basically a Q-homomorphism from Q^m to Q^n.

For Q-semimodules M, N, a function $\_\alpha : M \to N$ is said to be a Q-homomorphism iff:

- for $m_1, m_2 \in M$, we have, $(m_1 \oplus m_2)\alpha = m_1\alpha \oplus m_2\alpha$
- for $q \in Q$ and $m \in M$, we have, $(q \odot m)\alpha = q \odot (m\alpha)$.

So the matrix $\langle H \rangle$ for a heap H is an endomorphism from Q^n to Q^n where $n = |H|$. Also, an identity matrix is an identity homomorphism.

Next, for left Q-semimodules M, N, P and Q-homomorphisms $\_\alpha : M \to N$ and $\_\beta : N \to P$, the composition $\_(\alpha \circ \beta) : M \to P$ can be given by matrix multiplication, $\overline{\alpha} \times \overline{\beta}$. The composition is associative. And it obeys the identity laws.

This makes the set $V_Q = \{Q^n | n \in \mathbb{N}\}$ with $\mathrm{Hom}(Q^m, Q^n) = \mathcal{M}_{m,n}(Q)$ a category. We worked in this category. But there might be other such categories worth exploring.

8 RELATED WORK

8.1 Heap Semantics for Linear Logic

Computational and operational interpretations of linear logic have been explored in several works, especially in Chirimar et al. [1996], Turner and Wadler [1999]. In Turner and Wadler [1999], the authors provide a heap-based operational interpretation of linear logic. They show that a call-by-name calculus enjoys the single pointer property, meaning a linear resource has exactly one reference while a call-by-need calculus satisfies a weaker version of this property, guaranteeing only the maintenance of a single pointer. This system considers only linear and unrestricted resources. We generalize this operational interpretation of linear logic to quantitative type theory by allowing resources to be drawn from an arbitrary semiring. We derive a quantitative version of the single pointer property in 5. We can develop a quantitative version of the weak single pointer property for call-by-need reduction but for this, we need to modify the typing rules to allow sharing of resources.

8.2 Combining dependent and linear types

Perhaps the earliest work studying the combination of linear and dependent types was proposed in the form of a categorical model by Bonfante et al. [2001] who were interested in characterizing

how a linear dependent type system should be designed. A year later, Cervesato and Pfenning [2002] proposed the Linear Logical Framework (LLF) that combined non-dependent linear types with dependent types. This paper spurred a number of publications, but most relevant are in the line of work which extend dependent types with Girard et al. [1992]'s and Dal Lago and Hofmann [2009]'s bounded linear types. For example, Dal Lago and Gaboardi [2011]'s dlPCF, a sound and complete system for reasoning about evaluation bounds of PCF programs. Dal lago and Petit [2012] also show that dlPCF can also be used to reason about call-by-value execution, and Gaboardi et al. [2013b] develop a similar system called DFuzz for analyzing differential privacy of queries involving sensitive information. In the same vein, Krishnaswami et al. [2015] show how to combine non-dependent linear types with dependent types by generalizing Benton [1995]'s linear/non-linear logic. However, it was Luo [2018] who developed the first fully dependent type theory for linear logic both in a commutative and a non-commutative formalization.

Quantitative type theory is a newcomer in this area of linear and dependent types.

8.3 Quantitative Type Theory

McBride [2016a] and Atkey [2018] generalize of Girard et al. [1992]'s bounded linear logic (due to Ghica and Smith [2014]) to use elements of a resource semiring to track the usage of variables. These systems use a typing judgement of the form: $x_1 :^{\rho_1} A_1, x_2 :^{\rho_2} A_2, \ldots, x_n :^{\rho_n} A_n \vdash a :^{\rho} A$, where ρ_is and ρ are elements of a semiring. Roughly speaking, this judgement means that using ρ_i copies of x_i of type A_i, with i varying from 1 to n, we get ρ copies of a of type A. In contrast, the typing judgement of our system can only describe the production of one copy of a (i.e. the case where $\rho = 1$). To express other quantities of a in our system, one must multiply the context by ρ.

This difference means that the McBride/Atkey systems type check more terms because they may take advantage of the context in which they will eventually appear. For example, when the current usage is 0, such as in types, then no usage checking occurs. In contrast, for irrelevant terms, our system checks that the term is correct in some context, but then discards this information.

While including ρ as part of the judgement is more expressive, it also produces a system that is brittle. McBride's system does not admit a substitution lemma as the general case requires the ability to subtract and divide resources; Atkey repaired this issue by restricting ρ to be either 0 or 1 and requiring a few more properties to hold. Furthermore, neither system includes sub-usaging.

Atkey proved the soundness of his system with respect to a denotational model, whereas our proofs use a heap-based operational model. As a result, our approach extends to non-normalizing languages (such as those with **type** : **type**).

8.4 Quantities as modalities

Orchard et al. [2019] introduced a system with notion of graded necessity modalities—here called usage modalities—in a practical programming language with usage polymorphism, indexed types, and the use of arbitrary semirings. However, their system does not have full dependent types. They show that usage modalities can be used to encode a large number of graded coeffects in the style of Gaboardi et al. [2016] and Brunel et al. [2014]. A coeffect captures how a context is used, which is dual to an effect, and thus, usage modalities are graded comonads rather than graded monads [Fujii et al. 2016], which capture effects.

Abel and Bernardy [Abel and Bernardy 2020] use a quantitative type system to provide an abstract view of modalities. Their type system is similar in structure to ours, but its features and requirements differ. It includes usage and predicative parametric polymorphism but lacks an extension to dependent types. Their system is also strongly normalizing. Furthermore, Abel and Bernardy define a relational interpretation for their system and use it to derive parametricity principles. One property that they derive from this logical relation is that irrelevant arguments

do not affect computation (or equality). Due to our inclusion of the **type** : **type** axiom, this proof technique is unavailable to us, so we must use more syntactic methods. On the other hand, this axiom does not play a major role in our proofs: we conjecture that our approach to quantitative dependent types would work equally well in normalizing type theories.

8.5 Irrelevance and dependent types

There are several approaches to adding irrelevant quantification to dependently-typed languages. Miquel [2001] first added "implicit" quantification to a Curry-style version of the extended Calculus of Constructions. In this system, only the relevant parts of the computation may be explicit in terms, everything else must be implicit. Implicit arguments are thus those that do not appear free in the body of their abstractions. Barras and Bernardo [2008] showed how to support decidable type checking by allowing type annotations and other irrelevant subcomponents to appear in terms. In this setting, irrelevant arguments must not be free in the erasure of the body of their abstractions. Mishra-Linger and Sheard [2008] extended this approach to pure type systems. More recently, Weirich et al. [2017] used these ideas as part of a proposal for a core language for Dependent Haskell.

McBride [2016b], further refined by Atkey [2018], proposed using the 0 element in quantitative type theory to represent irrelevant quantification in dependent types. We have followed their design in making the usage of irrelevant variables in the co-domain of Π-types unrestricted in our system.

9 FUTURE WORK AND CONCLUSIONS

Quantitative Type Theory is a generic framework for expressing the flow and usage of resources in programs. In this work, we provide a new way of incorporating this framework into dependently-typed languages, with the goal of supporting both type erasure and linearity in a common system. In this extension, we get among other things, a new way to look at irrelevance in dependent type theory.

An ordinary, substitution-based operational semantics does not have the ability to model the use of resources represented by variables. Therefore, we use a heap semantics to model usage in our quantitative dependently-typed language. The heap semantics carries out all accounting of resources during evaluation of terms. We show that the type system is sound with respect to this heap semantics. Furthermore, since our heap semantics is bisimilar to the ordinary semantics, we show that the resource agnostic way of evaluation is sound with respect a resource aware semantics. So a Quantitative Type System is essentially a tool for static program analysis.

There are several things we would like to explore in future, some of which have been stated already. Can our heap semantics be used to derive parametricity properties using step-indexed logical relations? What happens when we add imperative features to our language like arrays? How would a general form of abstract equality upto erasure look like? What happens when we add multiple different modalities to our language?

The answers to these questions may have practical implications. Currently, languages such as Haskell, Rust, Idris, etc are experimenting with dependent and linear types, as well as the more general applications of QTT. Theoretical work in this direction provides guidance in these language designs and also resolves some practical problems for such programming languages.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). To appear.

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:29)2012

Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*. https://doi.org/10.1145/3209108.3209189

H. P. Barendregt. 1993. *Lambda Calculi with Types*. Oxford University Press, Inc., USA, 117–309.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures (FOSSACS 2008)*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Budapest, Hungary, 365–379.

P. N. Benton. 1995. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL '94)*. Springer-Verlag, London, UK, UK, 121–135.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Guillaume Bonfante, François Lamarche, and Thomas Streicher. 2001. *A model of a dependent linear calculus*. Intern report A01-R-262 || bonfante01c.

Edwin Brady. 2020. Idris 2: Quantitative Type Theory in Action. (Feb. 2020). Draft available from https://www.type-driven.org.uk/edwinb/idris-2-quantitative-type-theory-in-action.html.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370.

Iliano Cervesato and Frank Pfenning. 2002. A Linear Logical Framework. *Information and Computation* 179, 1 (2002), 19 – 75.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. 1996. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming* 6, 2 (March 1996), 195–244. https://doi.org/10.1017/S0956796800001660

U. Dal Lago and M. Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 133–142.

Ugo Dal Lago and Martin Hofmann. 2009. Bounded Linear Logic, Revisited. In *Typed Lambda Calculi and Applications*, Pierre-Louis Curien (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

Ugo Dal lago and Barbara Petit. 2012. Linear Dependent Types in a Call-by-Value Scenario. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. Association for Computing Machinery, New York, NY, USA, 115–126.

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg. 2018. Quantifiers for Dependent Haskell. GHC Proposal #102. https://github.com/goldfirere/ghc-proposals/blob/pi/proposals/0000-pi.rst

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–530.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013a. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013b. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 357–370.

Marco Gaboardi, Shin-ya Katsumata, Dominic A Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading.. In *ICFP*. 476–489.

Dan R Ghica and Alex I Smith. 2014. Bounded linear types in a resource semiring. In *European Symposium on Programming Languages and Systems*. Springer, 331–350.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66.

Jonathan S. Golan. 1999. *Semirings and their Applications*. Springer Netherlands. https://doi.org/10.1007/978-94-015-9333-5

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 17–30.

John Launchbury. 1993. A natural semantics for lazy evaluation. *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languagesl* (3 1993), 144–154. https://doi.org/10.1145/158511.158618

Z. Luo. 2018. Substructural Calculi with Dependent Types. *To appear at: The LINEARITY Workshop and Trends in Linear Logic and Applications* (2018).

Conor McBride. 2016a. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Conor McBride. 2016b. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233.

Alexandre Miquel. 2001. *The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping*. Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359. https://doi.org/10.1007/3-540-45413-6_27

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures (FoSSaCS)*. Springer.

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2020. Graded Modal Dependent Type Theory (Extended Abstract). *TyDe* (May 2020).

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 779–788. https://doi.org/10.1145/3209108.3209119

Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP, Article 32 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110276

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*. IEEE Computer Society, Washington, DC, USA, 221–. http://dl.acm.org/citation.cfm?id=871816.871845

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/1863543.1863568

David N. Turner and Philip Wadler. 1999. Operational interpretations of linear logic. *Theoretical Computer Science* 227, 1 (1999), 231 – 248. https://doi.org/10.1016/S0304-3975(99)00054-7

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2–3 (Jan. 1996), 167–187.

Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. 347–359.

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (July 2019), 29 pages. https://doi.org/10.1145/3341705

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

James Wood and Robert Atkey. 2020. A Linear Algebra Approach to Linear Metatheory. arXiv:cs.PL/2005.02247