A Graded Dependent Type System with a Usage-Aware Semantics

PRITAM CHOUDHURY, University of Pennsylvania, USA
HARLEY EADES III, Augusta University, USA
RICHARD A. EISENBERG, Tweag I/O, France and Bryn Mawr College, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

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

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

Additional Key Words and Phrases: Irrelevance, linearity, quantitative reasoning, heap semantics.

ACM Reference Format:
Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (January 2021), 32 pages. https://doi.org/10.1145/3434331

1 INTRODUCTION

Consider this typing judgement.

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

Here, the numbers in the context indicate 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 judgement of a *graded* type system which ensures that the *grades* or *quantities* annotating each in-scope variable reflects how it is used at run time. Graded type systems have been explored in much detail in the literature [Atkey 2018; Brunel et al. 2014; Gaboardi et al. 2016; Ghica and Smith 2014; McBride 2016; Orchard et al. 2019; Petricek et al. 2014]. The process of tracking usage through grades is straightforward, but this is a powerful method of instrumenting type systems with analyses of irrelevance and linearity that have practical benefits like erasure of irrelevant terms (resulting in speed-up) and compiler optimizations (such as in-place update

Authors' addresses: Pritam Choudhury, Computer and Information Science, University of Pennsylvania, USA, pritam@seas.upenn.edu; Harley Eades III, School of Computer and Cyber Sciences, Augusta University, 2500 Walton Way, Augusta, GA, 30904, USA, harley.eades@gmail.com; Richard A. Eisenberg, Tweag I/O, Paris, France, Computer Science, Bryn Mawr College, 101 N. Merion Ave, Bryn Mawr, PA, 19010, USA, rae@richarde.dev; Stephanie Weirich, Computer and Information Science, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA, sweirich@cis.upenn.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/1-ART50
https://doi.org/10.1145/3434331

of linear resources). This approach is also versatile. By abstracting over a domain of resources, the same form of type system can be used to guarantee safe memory usage, or prevent insecure information flow, or quantify information leakage, or identify irrelevant computations, or combine various modal logics. Several research languages, such as Idris 2 [Brady 2020] and Agda [Agda-Team 2020], are starting to adopt ideas from this domain, and new systems like Granule [Orchard et al. 2019] are being developed to explore its possibilities.

Our concrete motivation for studying graded type systems 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*: the compiler must be able to eliminate type arguments to polymorphic functions. Type erasure is key both to support parametric polymorphism and to efficiently execute Haskell programs. We discuss this in more detail in Section 2.

Although Haskell is our eventual goal, our work remains general. Our designs are compatible with the current approaches in GHC, but are not specialized to Haskell.

We make the following contributions in this paper:

- Our system flexibly abstracts over an algebraic structure used to count resources. Section 3 describes this structure—a partially-ordered semiring—and its properties. This use of a resource algebra is standard, although we identify subtle differences in its specification.
- Section 4 presents a simple graded type system, with standard algebraic types and a graded modal type. This system is not novel; instead, it establishes a foundation for the dependent system. However, even at this stage, we identify subtleties in the design space.
- Because the standard operational semantics does not track resources, type safety does not imply that usage tracking is correct. Section 5 describes a heap-based operational semantics, inspired by Turner and Wadler [1999]. Every variable in the heap has an associated resource tag from our abstract structure, modelling how resources are used during computation. We prove that our type system is sound with respect to this instrumented semantics. This theorem 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.
- Using soundness, we show (a generalization of) the *single pointer property* for linear resources in Section 6. The single pointer property says that a linear resource is referenced by precisely one pointer at runtime. Such a property would enable in-place updates of linear resources.
- Our key contribution is the design of the language, GRAD, extending our ideas to dependent types. In contrast to other approaches [Atkey 2018], we use the same rules to check relevant and irrelevant phrases (that is, terms and types). When computing the resources used by the entire term, we discard irrelevant usages. Types are irrelevant to computation, so our system ignores these usages. We describe the design of the type system in Section 7 and extend the soundness proof with respect to a heap semantics in Section 8.
  Our system is thus both simpler and more uniform than prior work that combines usage tracking with dependent types. In particular, Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016] disables resource checking in types, leading to limitations on the sorts of reasoning that can be done in the type system. On the other hand, Resourceful Dependent Types [Abel 2018] and GrTT [Moon et al. 2020] maintain separate counts of usages in types and terms, incurring additional bookkeeping for less benefit. Section 9.3 provides a detailed comparison of our work with QTT.

- We have mechanized, in Coq, some intricate syntactic properties of our development (substitution, weakening, preservation, progress). These proof scripts are available online at http://www.github.com/sweirich/graded-haskell.

An extended version of this paper is available at http://arxiv.org/abs/2011.04070.

2 OUR GOAL

While the exploration of graded type systems in this paper is applicable to a wide array of examples (see Section 3.2), we were originally motivated to study such systems in the context of GHC/Haskell, where we wish to combine its existing support for linearity [Bernardy et al. 2018] (available as of GHC 9.0) with support for dependent types [Eisenberg 2016; Gundry 2013; Weirich et al. 2019, 2017].

A key advantage of a successful combination of graded and dependent types for Haskell is that it allows us to use the 0 quantity to mean *irrelevant* usage, where an irrelevant sub-term is one that is not needed while computing the reduct of the concerned term. Irrelevant sub-terms are quite common in terms derived in dependent type systems. They are essential for type-checking the terms but if left as such, they can make programs run much slower. So we need to track irrelevant sub-terms and erase them before running a program. Weirich et al. [2017] use a relevance tag $+/-$ on the Π for this purpose. On the other hand, Bernardy et al. [2018] use a linearity tag $1/\omega$ on the function domain type to track linear usage. We can combine these two together using 0, 1 and ω to track irrelevant, linear and unrestricted usages respectively. This has an added advantage. It will allow us to provide Haskell programmers the option of annotating arguments with a usage tag that subsumes relevance and linearity tags [Eisenberg 2018]. So the use of 0 to mark irrelevance fits in swimmingly with Haskell's current story around linear types.

Furthermore, given that we plan to implement these ideas concretely inside GHC, it is essential that the system be as simple as possible. As discussed in more detail in Section 9.3, our system eliminates features that are not necessary in our case. Doing so will aid in integration with the rest of the GHC implementation.

Our intentions laid out, we start our exploration by reviewing semirings, the key algebraic structure used to abstractly represent grades.

3 THE ALGEBRA OF QUANTITIES

The goal of a graded 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 evaluation 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; Miquel 2001; Pfenning 2001; Reed and Pierce 2010; Volpano et al. 1996].

This generality derives from the fact that the type system is parametrized over an abstract algebraic structure of *grades* to model resources.[1] The abstract algebraic structure enables addition and multiplication of resources and these operations conform to our general understanding of resource arithmetic. A partially-ordered semiring is one such algebraic structure that captures this idea of resource modelling nicely.

[1] Grades are also called quantities, modalities, resources, coeffects or usages.

3.1 Partially-Ordered Semirings

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 and 1, 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 drawn from a semiring 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 higher demand than is actually necessary. For example, we may need to use some variable only once but it may be convenient to declare that the usage of that variable is 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, if $q_1 \leq q_2$, then for any $q \in Q$, we should have $q + q_1 \leq q + q_2$, $q \cdot q_1 \leq q \cdot q_2$, 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. In practice, it means our design is applicable to settings beyond the simple use of natural numbers to count resources.

3.2 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 drawing quantities from this semiring distinguishes between variables that are used (marked with one) and ones that are unused (marked with zero). In such a system, the quantity 1 does *not* correspond to a linear usage: this system does not count usage, but instead checks *whether* a variable is used or not.

There are two different partial orders that make sense for the boolean semiring. If we use the reflexive relation, then this type system tracks relevance precisely. 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 $1 + 1 = \omega$, $\omega + 1 = \omega$, and $\omega \cdot \omega = \omega$. A system using the linearity semiring tracks linearity by marking linear variables with 1 and unrestricted variables with ω. A suitable ordering in this semiring is the reflexive closure of $\{(0, \omega), (1, \omega)\}$. We do *not* want $0 \leq 1$, since then we would not 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) with irrelevance (0) and unrestricted usage (ω).

The *five-point linearity 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",

(Grammar)

types	A, B	$::=$	$\mathbf{Unit} \mid {}^{q}A \to B \mid \Box^{q} A \mid A \otimes B \mid A \oplus B$
terms	a, b	$::=$	$x \mid \lambda x{:}^{q} A.a \mid a\ b$
		$\mid$	$\mathbf{unit} \mid \mathbf{let\ unit}\ =\ a\ \mathbf{in}\ b \mid \mathbf{box}_{q}\ a \mid \mathbf{let\ box}\ x\ =\ a\ \mathbf{in}\ b$
		$\mid$	$(a, b) \mid \mathbf{let}\ (x, y) = a\ \mathbf{in}\ b$
		$\mid$	$\mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}_{q}\ a\ \mathbf{of}\ b_1; b_2$
usage contexts	Γ	$::=$	$\varnothing \mid \Gamma, x{:}^{q} A$
contexts	Δ	$::=$	$\varnothing \mid \Delta, x{:}A$
typing judgement			$\boxed{\Delta\ ;\ \Gamma \vdash a : A}$

Fig. 1. The simply typed graded λ-calculus

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.

A *security semiring* is based on a lattice of security levels, with increasing order representing decreasing security. The $+$ and $\cdot$ correspond to the join and meet operations of the lattice respectively. The partial order corresponds to the lattice order and 0 and 1 are the Private and Public security levels respectively. Public can never be as or more secure than Private, i.e. Public $\not\leq$ Private. This lattice may include additional elements besides Private and Public, corresponding to multiple levels of secrecy. As Abel and Bernardy [2020] describe, security type systems defined in this way differ from the usual convention (such as Abadi et al. [1999]) in that security levels are relative to 1, the level of the program under execution.

Many other examples of semirings are possible. Orchard et al. [2019] and Abel and Bernardy [2020] include comprehensive lists of several other applications including a type system for differential privacy [Reed and Pierce 2010] and a type system that tracks covariant/contravariant use of assumptions.

Partially-ordered semirings have been used to track resource usage in many type systems[Abel and Bernardy 2020; Atkey 2018; Brunel et al. 2014; Gaboardi et al. 2016; Ghica and Smith 2014; McBride 2016; Orchard et al. 2019; Petricek et al. 2014] but there are some variations with respect to the formal requirements. For example: Brunel et al. [2014] require the underlying set (of the semiring) along with the order to form a bounded sup-semilattice while Abel and Bernardy [2020] define the order using an additional meet operation on the underlying set; McBride [2016] uses a hemiring (a semiring without 1) while Atkey [2018] uses a semiring where zero-usage satisfies a certain condition. Our theory is parametrized by a partially ordered semiring as defined in Section 3.1. We add additional constraints as required only while deriving specific properties in Section 6.

4 A SIMPLE GRADED 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 similar to the system of Petricek et al. [2014]. The grammar for this system appear in Figure 1 on page 5. It is parametrized over an arbitrary partially-ordered semiring $(Q, 1, \cdot, 0, +, \leq)$ with grades $q \in Q$.

The typing judgement for this system has the form $\Delta\ ;\ \Gamma \vdash a : A$; the rules appear inline below. This judgement includes both a standard typing context Δ and a usage context Γ, a copy of the typing context annotated with grades. For brevity in examples, we often elide the standard typing

context as the information is subsumed by the usage context. Indeed, in any derivation, the typing context and the usage context correspond:

Notation 4.1.

- *The notation* $\lfloor \Gamma \rfloor$ *denotes a typing context* Δ *same as* Γ*, but with no grades.*
- *The notation* $\overline{\Gamma}$ *denotes the vector of grades in* Γ*.*
- *The notation* $\Delta \vdash \Gamma$ *denotes that* $\Delta = \lfloor \Gamma \rfloor$*.*

Lemma 4.2 (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: we could have only one usage context Γ and add constraints, or have a typing context Δ and a separate vector of quantities.

4.1 Type System Basics

We are now ready to start our tour of the typing rules of this system.

Variables.

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

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 systems, the context is extended to include $0 \cdot \Gamma$: this notation means that all variables in Γ must have a quantity of 0.

Notation 4.3 (Context scaling). *The notation* $q \cdot \Gamma$ *denotes a context* Γ' *such that, for each* $x{:}^r A \in \Gamma$*, we have* $x{:}^{q \cdot r} A \in \Gamma'$*.*

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.

Sub-usage.

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

We may allow our contexts to provide more resources than is necessary. Sub-usaging, as it is commonly referred to, allows us to assume more resources in our context than are necessary.

Notation 4.4 (Context sub-usage). *The notation* $\Gamma_1 \leq \Gamma_2$ *means* $\lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor$ *where, for every corresponding pair of assumptions* $x{:}^{q_1} A \in \Gamma_1$ *and* $x{:}^{q_2} A \in \Gamma_2$*, the condition* $q_1 \leq q_2$ *holds.*

Functions.

$$\text{ST-Lam}\ \frac{\Delta, x{:}A \,;\, \Gamma, x{:}^q A \vdash a : B}{\Delta \,;\, \Gamma \vdash \lambda x{:}^q A.a : ({}^q A \to B)} \qquad \text{ST-App}\ \frac{\Delta \,;\, \Gamma_1 \vdash a : ({}^q A \to B) \qquad \Delta \,;\, \Gamma_2 \vdash b : A}{\Delta \,;\, \Gamma_1 + q \cdot \Gamma_2 \vdash a\, b : B}$$

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 this rule,

we see that the function a has type ${}^{q}A \to B$, 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 can be used q times. Put another way, we must *multiply* the usage required for b, as recorded in the typing context Γ_2, 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 4.5 (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 obtained by point-wise addition of quantities; i.e. for every $x :^{q_1} A \in \Gamma_1$ and $x :^{q_2} A \in \Gamma_2$, we have $x :^{q_1+q_2} A \in \Gamma_3$.*

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 4.6 (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 ${}^{0}B \to {}^{1}A \to A$. (Just from this type, we can see that f must be constant in B.) Suppose also that we want to apply this function to some variable x. In this case, define the usage contexts

$$\Gamma_0 = f :^{1} ({}^{0}B \to {}^{1}A \to A) \qquad \Gamma_1 = \Gamma_0, x :^{0} B \qquad \Gamma_2 = f :^{0} ({}^{0}B \to {}^{1}A \to A), x :^{1} B$$

and construct a typing derivation for the application:

$$\text{ST-App}\ \frac{\text{ST-Weak}\ \dfrac{\text{ST-Var}\ \dfrac{}{\lfloor \Gamma_0 \rfloor\ ;\ \Gamma_0 \vdash f : {}^{0}B \to {}^{1}A \to A}}{\lfloor \Gamma_1 \rfloor\ ;\ \Gamma_1 \vdash f : {}^{0}B \to {}^{1}A \to A} \qquad \text{ST-Var}\ \dfrac{}{\lfloor \Gamma_1 \rfloor\ ;\ \Gamma_2 \vdash x : B}}{\lfloor \Gamma_1 \rfloor\ ;\ \Gamma_1 + 0 \cdot \Gamma_2 \vdash f\ x : {}^{1}A \to A}$$

Working through the context expression $\Gamma_1 + 0 \cdot \Gamma_2$, we see that the computed final context, derived in the conclusion of the application rule is just Γ_1 again. Although the variable x appears free in the expression $f\ x$, because it is the argument to a constant function here, this use does not contribute to the overall result.

4.2 Data Structures

Unit.

$$\text{ST-Unit}\ \frac{}{\varnothing\ ;\ \varnothing \vdash \mathbf{unit} : \mathbf{Unit}} \qquad \text{ST-UnitE}\ \frac{\Delta\ ;\ \Gamma_1 \vdash a : \mathbf{Unit} \quad \Delta\ ;\ \Gamma_2 \vdash b : B}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash \mathbf{let\ unit}\ =\ a\ \mathbf{in}\ b : B}$$

The **Unit** type has a single element, **unit**. To eliminate a term of this type, we just match it with **unit**. Since the elimination form requires the resources used for both the terms, we add the two contexts in the conclusion of rule ST-UnitE.

The graded modal type.

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

The type $\Box^q A$ is called a *graded modal type* or *usage modal type*. It is introduced by the construct $\mathbf{box}_q\ a$, which uses the expression q times to build the box. This box can then be passed around as an entity. When unboxed (rule ST-LetBox), the continuation has access to q copies of the contents.

Products.

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

The type system includes (multiplicative) products, also known as tensor products. The two components of these pairs do not share variable usages. Therefore the introduction rule adds the two contexts together. These products must be eliminated via pattern matching because both components must 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.

$$\text{ST-Inj1}\ \frac{\Delta\ ;\ \Gamma \vdash a : A_1}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_1\ a : A_1 \oplus A_2} \qquad \text{ST-Inj2}\ \frac{\Delta\ ;\ \Gamma \vdash a : A_2}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_2\ a : A_1 \oplus A_2}$$

$$\text{ST-Case}\ \frac{1 \le q \qquad \Delta\ ;\ \Gamma_1 \vdash a : A_1 \oplus A_2 \qquad \Delta\ ;\ \Gamma_2 \vdash b_1 : {}^qA_1 \to B \qquad \Delta\ ;\ \Gamma_2 \vdash b_2 : {}^qA_2 \to B}{\Delta\ ;\ q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B}$$

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 can 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 the scrutinee at least once, as indicated by the $1 \le q$ constraint.

4.3 Type Soundness

For the language presented above, we define an entirely standard call-by-name reduction relation $a \rightsquigarrow a'$, included in the extended version of this paper. With this operational semantics, a syntactic proof of type soundness follows in the usual manner, via the entirely standard progress and preservation lemmas. The substitution lemma that is part of this proof is of particular interest to us, as it must account for the number of times the substituted variable (x, in our statement) is used when computing the contexts used in the conclusion of the lemma:

Lemma 4.7 (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* $\Delta_1, \Delta_2\ ;\ \Gamma_1 + q \cdot \Gamma, \Gamma_2 \vdash b\{a/x\} : B$.

4.4 Discussion and Variations

At this point, the language that we have developed recalls systems found in prior work, such as Brunel et al. [2014], 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] include a slightly more expressive form of pattern matching. Their elimination forms for the box modality and products multiply each scrutinee by a 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] require that the semiring include least-upper bounds for the partial order of the semiring. This allows them to compose case branches with differing usages.
- Third, in the rule for **case**, like Abel and Bernardy [2020], we need the requirement that $1 \leq q$. In our system as well as theirs, it turns out that this requirement is not motivated by the standard type soundness theorem: 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.

The standard type soundness theorem is not very informative because it does not show that the quantities are correctly used. Therefore, to address this issue, we turn to a heap-based semantics, based on Launchbury [1993] and Turner and Wadler [1999], to account for resource usage during computation.

5 HEAP SEMANTICS FOR SIMPLE TYPE SYSTEM

A heap semantics shows how a term evaluates when the free variables of the term are assigned other 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:[2]

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

5.1 The Step Judgement

The reduction above is expressed informally as a sequence of pairs of heap H and expression a. We formalize this relation using the following judgement, which appears in Fig 2.

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

The meaning of this relation is that r copies of the term a use the resources of the heap H and step to r copies of the term a', with H' being the new heap. The relation also maintains additional information, which we explain below.

Heap assignments are of the form $x \overset{q}{\mapsto} \Gamma \vdash a : A$, associating an *assignee variable* with its *allowed usage* q and assignment a. The *embedded context* Γ and type A help in the proof of our soundness theorem (5.11). For a heap H, we use $\lfloor H \rfloor$ to represent H excluding the allowed usages and $\lVert H \rVert$ to represent just the list of underlying assignments. We call $\lfloor H \rfloor$ and $\lVert H \rVert$ the *erased* and *bare* views

[2]We don't have **Int** type and + function in our language, but we use them for the sake of explanation.

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

$$\textsc{Small-Var}\ \frac{1 \le r}{[H_1, x \overset{(q+r)}{\mapsto} \Gamma \vdash a : A, H_2]\ x \Rightarrow^{r}_{S} [H_1, x \overset{q}{\mapsto} \Gamma \vdash a : A, H_2\,;\, \mathbf{0}^{|H_1|} \diamond r \diamond \mathbf{0}^{|H_2|}\,;\, \varnothing]\ a}$$

$$\textsc{Small-AppBeta}\ \frac{x \notin \mathrm{Var}\, H \cup \mathrm{fv}\, b \cup \mathrm{fv}\, a - \{y\} \cup S \qquad a' = a\{x/y\}}{[H]\ (\lambda y{:}^{q} A_1.a)\ b \Rightarrow^{r}_{S} [H, x \overset{r\cdot q}{\mapsto} \Gamma \vdash b : A\,;\, \mathbf{0}^{|H|} \diamond 0\,;\, x{:}^{r\cdot q} A]\ a'}$$

$$\textsc{Small-AppL}\ \frac{[H]\ a \Rightarrow^{r}_{S \cup \mathrm{fv}\, b} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'}{[H]\ a\ b \Rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'\ b}$$

$$\textsc{Small-CaseL}\ \frac{[H]\ a \Rightarrow^{r\cdot q}_{S \cup \mathrm{fv}\, b_1 \cup \mathrm{fv}\, b_2} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'}{[H]\ \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 \Rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ \mathbf{case}_q\ a'\ \mathbf{of}\ b_1; b_2}$$

$$\textsc{Small-Case1}\ \frac{}{[H]\ \mathbf{case}_q\ (\mathbf{inj}_1\ a)\ \mathbf{of}\ b_1; b_2 \Rightarrow^{r}_{S} [H\,;\,\mathbf{0}^{|H|}\,;\,\varnothing]\ b_1\ a}$$

$$\textsc{Small-Case2}\ \frac{}{[H]\ \mathbf{case}_q\ (\mathbf{inj}_2\ a)\ \mathbf{of}\ b_1; b_2 \Rightarrow^{r}_{S} [H\,;\,\mathbf{0}^{|H|}\,;\,\varnothing]\ b_2\ a}$$

$$\textsc{Small-Sub}\ \frac{[H_1]\ a \Rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a' \qquad H_1 \le H_2}{[H_2]\ a \Rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ a'}$$

Fig. 2. Heap semantics (excerpt)

Notation 5.1. *The notation $\mathbf{0}^n$ denotes a vector of 0's of length n. The notation $\mathbf{u}_1 \diamond \mathbf{u}_2$ denotes concatenation. Here fv a stands for the free variables of a while* Var *H stands for the domain of H and the free variables of the terms appearing in the assignments of H.*

of H respectively. For example, for $H = [x \overset{q}{\mapsto} \Gamma \vdash a : A]$, the erased view $\lfloor H \rfloor = [x \mapsto \Gamma \vdash a : A]$ and the bare view $\lVert H \rVert = [x \mapsto a]$ and Γ is the embedded context. The vector of allowed usages of the variables in H is denoted by $\overline{H}$.

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. 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 allowed usages of their assignments using the added context Γ'.[3] Therefore, after a step $[H]\ a \Rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, 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

[3]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.

$q = q' + 1$. 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 $\mathbf{u}'$ called *consumption vector*, with its components showing usage at the corresponding variables in H'. (The length of $\mathbf{u}'$ will always be the same as H'.)

Finally, owing to the presence of **case** expressions that can use the scrutinee more than once, we need to be able to evaluate several copies of the scrutinee in parallel before passing them on to the appropriate branch. So we step r copies of a term a in parallel to get r copies of a'. We call r the *copy quantity* of the step. For the most part, we shall be interested in copy quantity of 1.

Notation 5.2. *We use* $[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$ *to denote* $[H]\ a \Rightarrow_S^1 [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$.

5.2 Reduction Relation

Figure 2 contains an excerpt of the reduction relation. They mirror the ordinary small-step rules, but there are some crucial differences. For example, this relation includes rule Small-Var that allows a variable look-up, provided its usage permits. The look-up consumes the copy quantity from the allowed usage. But the copy quantity cannot be arbitrary here – we are consuming the resource at least once. So we restrict the copy quantity to be 1 or more. This is the only rule that modifies the usage of an existing variable in the heap.

In this relation, rule AppBeta loads new assignment into the heap instead of immediate substitution. The substitution happens in steps through variable look-ups. To avoid conflict, we choose new variables excluding the ones already in use. Since we are evaluating r copies, we set the allowed usage of the variable to $r \cdot q$ where q is the usage annotation on the term.

Let us look at rule Small-CaseL. This is interesting since the copy quantity in the premise and the conclusion are different. In fact, we introduced copy quantity to properly handle usages while evaluating **case** expressions. For evaluating r copies of the **case** expression, we need to evaluate $r \cdot q$ copies of the scrutinee since the scrutinee gets used q times in either branch.

The rule Small-Sub reduces the allowed usages in the heap and then lets the term take a step. Here, we use $H_1 \leq H_2$ to mean $\lfloor H_1 \rfloor = \lfloor H_2 \rfloor$ and for corresponding pair of assignments $x \overset{q_1}{\mapsto} \Gamma \vdash a : A$ and $x \overset{q_2}{\mapsto} \Gamma \vdash a : A$ in H_1 and H_2 respectively, the condition $q_1 \leq q_2$ holds.

The multi-step reduction relation is the transitive closure of the single-step relation. In rule Multi-Many, the consumption vectors from the steps are added up and the added contexts of new variables are concatenated. The copy quantity is the same in both the premises and the conclusion since the rule represents parallel multi-step evaluation of r copies.

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

Multi-One

$$\frac{[H]\ a \Rightarrow_S^r [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ b}{[H]\ a \Rightarrow_S^r [H'\,;\,\mathbf{u}'\,;\,\Gamma]\ b}$$

Multi-Many

$$\frac{[H]\ a \Rightarrow_S^r [H'\,;\,\mathbf{u}'\,;\,\Gamma_1]\ b_1 \qquad [H']\ b_1 \Rightarrow_S^r [H''\,;\,\mathbf{u}''\,;\,\Gamma_2]\ b}{[H]\ a \Rightarrow_S^r [H''\,;\,\mathbf{u}' \diamond \mathbf{0}^{|\Gamma_2|} + \mathbf{u}''\,;\,\Gamma_1, \Gamma_2]\ b}$$

5.3 Accounting of Resources

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

Theorem 5.3 (Conservation). *If* $[H]\ a \Rightarrow\!\!\!\!\!\rightarrow^{r}_{S} [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$, *then* $\overline{H'} + \mathbf{u}' \leq \overline{H} \diamond \overline{\Gamma'}$.

Here, $\overline{H}$ represents the initial 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 concatenated with those that are added during evaluation, are equal to or more than the remaining resources plus those that were used up. Note that if the partial order is the trivial reflexive order, $\leq$ becomes an equality. In such a scenario, the reduction relation enforces strict conservation of resources. More generally, this theorem states that we don't use more resources than what we are entitled to.

Unlike the substitution-based semantics, in this heap semantics, terms can "get stuck" due to lack of resources. Let us look at the following evaluation:

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

The evaluation 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.

But this is not the only way in which an evaluation can run out of resources. Such a situation may also happen through "unwise usage", even when the starting heap contains enough resources. For example, over the linearity semiring, the evaluation: $[x \overset{\omega}{\mapsto} 5]x + (x + x) \Rightarrow [x \overset{1}{\mapsto} 5]5 + (x + x) \Rightarrow [x \overset{0}{\mapsto} 5]5 + (5 + x)$ gets stuck because in the first step, ω was "unwisely" split as $1 + 1$ instead of being split as $\omega + 1$.

Our aim, then, is to show that given a heap that contains enough resources, a well-typed term that is not a value, can always take a step such that the resulting heap contains enough resources for the evaluation of the resulting term. We shall formalize what it means for a heap to contain enough resources. But before that, let us explore the relationship between the various possible steps a term can take when provided with a heap.

5.4 Determinism and Alpha-Equivalence

Earlier, we pointed out that the step relation is non-deterministic. But on a closer look, we find that the non-determinism is limited more or less to the usages. If a term steps in two different ways when provided with a heap, the resulting terms are the same; the resulting heaps, though, may have different allowed usage vectors. Here, we formulate a precise version of this statement.

A term, when provided with a heap, can step either by looking up a variable or by adding a new assignment. Now, if the heap does not contain duplicate assignments for the same variable, look-up will always produce the same result. We call such heaps *proper*. Note that the reduction relation maintains this property of heaps. So hereafter, we restrict our attention to proper heaps. Next, if a term steps by adding a new assignment, we may choose different fresh variables leading to different resulting terms. But such a difference is reconcilable. Viewed as closures, such heap term pairs are α-equivalent.

Given a heap H and a term a, let us call $(\lVert H \rVert, a)$ a *machine configuration*. Two heap term pairs are α-equivalent if the corresponding machine configurations are identical up to systematic renaming of assignee variables. We denote α-equivalence by $\sim_\alpha$.

The step relation, then, is deterministic in the following sense:

Lemma 5.4 (Determinism). *If* $[H_1]\ a_1 \Rightarrow_S^{r_1} [H_1' ; \mathbf{u}_1' ; \Gamma_1']\ a_1'$ *and* $[H_2]\ a_2 \Rightarrow_S^{r_2} [H_2' ; \mathbf{u}_2' ; \Gamma_2']\ a_2'$ *and* $(H_1, a_1) \sim_\alpha (H_2, a_2)$, *then* $(H_1', a_1') \sim_\alpha (H_2', a_2')$.

The lemma above says that, if a term, when provided with a heap, takes a step in two different ways, then the resulting heap term pairs are basically the same. We know that the ordinary small-step semantics is deterministic. The inclusion of allowed usages in the heap semantics is not to produce multiple reducts but just to block evaluation at the point where consumption reaches its permitted limit. This is an important point and needs elaboration.

Call a reduction consisting of n steps an n-chain reduction. Also, for a reduction $[H]\ a \Rightarrow_S^r [H' ; \mathbf{u}' ; \Gamma']\ a'$, call $[\lVert H \rVert]a \Rightarrow [\lVert H' \rVert]a'$ the *machine view* of reduction. Then, the machine view of every n-chain reduction of a term in a heap is the same, modulo α-equivalence. So, if there *exists* an n-chain reduction of a to a', starting with heap H, we know that there is a way by which a can reduce to a' without running out of resources, implying the validity of the reduction. In such a scenario, we may as well forget all the usage annotations and evaluate a for n steps starting with $\lVert H \rVert$. By the above lemma, such an evaluation in this machine environment is deterministic and hence unique. The reduced heap term pair that we get is the same (modulo α-equivalence). Along with the soundness theorem, this shall give us a deterministic reduction strategy that is correct.

Now that we see the equivalence of all the possible reducts, we explore its relation with the ordinary small-step reduct.

5.5 Bisimilarity

The ordinary and the heap-based reduction relations are bisimilar in a way we make precise below. To compare, 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 variable appearing subsequently in the heap. Note that the reduction relation preserves acyclicity. Hereafter, we restrict our attention to proper, acyclic heaps.

Now, for a heap H, define $a\{H\}$ as the term obtained by substituting in a, in reverse order, the corresponding terms for the variables in the heap. Then we have the following lemmas:

Lemma 5.5. *If* $[H]\ a \Rightarrow_S^r [H' ; \mathbf{u}' ; \Gamma']\ a'$, *then* $a\{H\} = a'\{H'\}$ *or* $a\{H\} \rightsquigarrow a'\{H'\}$. *Further, if* $[\varnothing]\ a \Rightarrow_S^r [H' ; \mathbf{u}' ; \Gamma']\ a'$, *then* $a \rightsquigarrow a'\{H'\}$.

Lemma 5.6. *If* $a \rightsquigarrow a_1$, *then for a heap* H, *we have* $[H]\ a \Rightarrow_S^r [H, H' ; \mathbf{u} ; \Gamma]\ a_2$ *where* $a_2\{H'\} = a_1$.

The heap reduction relation splits the ordinary β-reduction rules into an assignment addition rule and a variable look-up rule. This enables the heap-based rules to substitute one occurrence of a variable at a time while the ordinary β-rules substitute all occurrences of a variable at once. If we perform substitution immediately after loading a new assignment to the heap, then the heap-based rules and the ordinary step rules are essentially the same. The above lemmas formalize this idea.

The heap-based rules substitute one occurrence of a variable at a time and keep track of usage and obstruct unfair usage. With this constraint in place, we ought to know how much resources shall be necessary before evaluating a term. This will tell us how much resources the starting heap should contain. The type system helps us know this as we see next.

5.6 Heap Compatibility

The key idea behind this language design is that, if the resources contained in a heap are judged to be "right" for a term by the type system, the evaluation of the term in such a heap does not get

stuck. With the heap-based reduction rules enforcing fairness of usage, this would mean that the type system does a proper accounting of the resource usage of terms.

The compatibility relation $H \vdash \Delta; \Gamma$ presented below expresses the judgement that the heap H contains enough resources to evaluate any term that type-checks in the usage context Γ. A heap that is compatible with some context is called a *well-formed* heap.

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

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

The rule Compat-Cons rule reminds us of the substitution lemma 4.7. In a way, this rule is converse of the substitution lemma. It loads q potential single-substitutions into the heap and lets the context use the variable q times.

Example 5.7. Consider the following derivation:[4]

$$\dfrac{\dfrac{\dfrac{\varnothing \vdash \varnothing \qquad \varnothing \vdash 1 : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1 \vdash x_1 :^7 \mathbf{Int}} \qquad x_1 :^2 \mathbf{Int} \vdash x_1 + x_1 : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1, x_2 \overset{3}{\mapsto} x_1 + x_1 \vdash x_1 :^1 \mathbf{Int}, x_2 :^3 \mathbf{Int}} \qquad x_1 :^1 \mathbf{Int}, x_2 :^2 \mathbf{Int} \vdash x_1 + (x_2 + x_2) : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1, x_2 \overset{3}{\mapsto} x_1 + x_1, x_3 \overset{1}{\mapsto} x_1 + (x_2 + x_2) \vdash x_1 :^0 \mathbf{Int}, x_2 :^1 \mathbf{Int}, x_3 :^1 \mathbf{Int}}$$

The context $x_1 :^7 \mathbf{Int}$ splits its resources amongst derivations of $x_2 = x_1 + x_1$ (thrice) and $x_3 = x_1 + (x_2 + x_2)$ (once). The heap keeps a record, in the form of allowed usages, of how the context gets split. A heap compatible with a context, therefore, satisfies the resource demands of a term derived in this context.

We pointed out earlier that the rule Compat-Cons is like a converse substitution lemma. The following lemma formalizes this idea:

Lemma 5.8 (Multi-substitution). *If* $H \vdash \Delta; \Gamma$ *and* $\Delta\ ;\ \Gamma \vdash a : A$, *then* $\varnothing\ ;\ \varnothing \vdash a\{H\} : A$.

Because rule Compat-Cons leads to expansion (or reverse substitution), we can re-substitute while maintaining well-typedness. The compatibility relation is crucial to our development. So we explore it in more detail below.

5.7 Graphical and Algebraic Views of the Heap

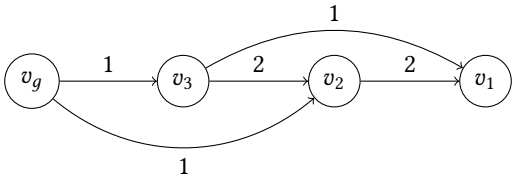
A heap can be viewed as a memory graph where the assignee variables correspond to memory locations and the assigned terms to data stored in those locations. The allowed usage then, is the number of ways the location can be referenced. This gives us a graphical view of the heap.

A 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. For $x_i :^{q_{ji}} A_i \in \Gamma_j$ (where $i < j$) add an edge with weight $w(v_j, v_i) := q_{ji}$ from v_j to v_i. (Note that Γ_j only contains variables

[4]For simplicity, we omit the Δs from the compatibility and the typing judgements.

x_1 through x_{j-1}.) 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 5.7, we have the following memory graph[5]:



For a heap compatible with some context, we can express the allowed usages of the assignee variables in terms of the edge weights of the memory graph. Let us define the length of a path to be the product of the weights along the path. Then, the allowed usage of a variable is the sum of the lengths of all paths from the source node to the node corresponding to that variable. 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 allowed usage of the assignment for v_j in the heap. This means that the allowed usage of an assignment is equal to the (static) count of the number of times the concerned location is referenced. So, we also call q_j the *count* of v_j and call this property count balance. Below, we present an algebraic formalization of this property of well-formed heaps.

Notation 5.9. *We use* $\mathbf{0}$ *to denote a row vector of* 0*s of length* n *(when* n *is clear from the context) and use* $\mathbf{0}^\intercal$ *to denote a column vector of* 0*s.*

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. The transformation matrix for example 5.7 is:

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

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 the acyclicity of the graph. With the matrix operations over a semiring defined in the usual way, the count balance property is:

Lemma 5.10 (Count Balance). *If* $H \vdash \Delta; \Gamma$, *then* $\overline{H} = \overline{H} \times \langle H \rangle + \overline{\Gamma}$.

Proof. We show this by induction on $H \vdash \Delta; \Gamma$. The base case is trivial.

For the Cons-case, let $H', x \overset{q}{\mapsto} \Gamma_2 \vdash a : A \vdash \Delta', x{:}A; \Gamma_1, x :^q A$ where $H' \vdash \Delta'; \Gamma_1 + (q \cdot \Gamma_2)$. By inductive hypothesis, $\overline{H'} = \overline{H'} \times \langle H' \rangle + \overline{\Gamma_1 + (q \cdot \Gamma_2)}$. Therefore, $\overline{H'} \diamond q = \overline{H'} \diamond q \times \begin{pmatrix} \langle H' \rangle & \mathbf{0}^\intercal \\ \overline{\Gamma_2} & 0 \end{pmatrix} + \overline{\Gamma_1} \diamond q$. □

For example 5.7, we can check that $\overline{H} = \begin{pmatrix} 7 & 3 & 1 \end{pmatrix}$ 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 neighbours and the weights of the corresponding edges. We have, $q_i = \Sigma_j \, q_j w(v_j, v_i) + w(v_g, v_i)$.

[5]We omit the 0 weight edge from v_g to v_1.

The right-hand side of this equation represents static estimate of demand, the amount of resources we shall need while the left-hand side represents 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.

Therefore, if $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$, we should be able to evaluate a in H without running out of resources. This is the gist of the soundness theorem.

5.8 Soundness

Theorem 5.11 (Soundness). *If $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$ and $S \supseteq \mathrm{dom}\,\Delta$, then either a is a value or there exists Γ', H', $\mathbf{u}'$, Γ_4 such that:*

- $[H]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ a'$
- $H' \vdash \Delta, \lfloor \Gamma_4 \rfloor; \Gamma'$
- $\Delta, \lfloor \Gamma_4 \rfloor\ ;\ \Gamma' \vdash a' : A$
- $\overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_4} \times \langle H' \rangle \leq \overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_4}$

The soundness theorem[6] 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, $[x \overset{1}{\mapsto} \mathbf{unit}]x \Rightarrow [x \overset{0}{\mapsto} \mathbf{unit}]\mathbf{unit}$ and $x :^1 \mathbf{Unit} \vdash x : \mathbf{Unit}$ while $x :^0 \mathbf{Unit} \vdash \mathbf{unit} : \mathbf{Unit}$. Though the typing context may change, the new context, which type-checks the reduct, must be compatible with the new heap. This means that we can apply the soundness theorem again and again until we reach a value. At every step of the evaluation, the dynamics of our language aligns perfectly with the statics of the language. Graphically, as the evaluation 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 according to the fourth clause of the theorem. For the moment being, let the partial order be the trivial reflexive order. Then, the equation stands as:

$$\begin{array}{rcccl} & \overline{\Gamma'} & + & \mathbf{u}' & + & \mathbf{0} \diamond \overline{\Gamma_4} \times \langle H' \rangle \\ = & \overline{\Gamma} \diamond \mathbf{0} & + & \mathbf{u}' \times \langle H' \rangle & + & \mathbf{0} \diamond \overline{\Gamma_4} \end{array}$$

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 $\mathbf{0} \diamond \overline{\Gamma_4}$ and gets $\mathbf{0} \diamond \overline{\Gamma_4} \times \langle H' \rangle$ 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.

For an arbitrary partial order, the transaction gets skewed in favour of the heap; meaning, the context gets less from the heap for what it pays. This is so because the heap contains more resources than is necessary; so it may "throw away" the extra resources.

This soundness theorem subsumes ordinary type soundness. In fact, we can derive the ordinary preservation and progress lemmas from this soundness theorem using bisimilarity of the two reduction relations and the multi-substitution property.

Corollary 5.12. *If $\varnothing\ ;\ \varnothing \vdash a : A$ and $a \rightsquigarrow b$, then $\varnothing\ ;\ \varnothing \vdash b : A$.*

Proof. Since $a \rightsquigarrow b$, for any S, we have, $[\varnothing]\ a \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma']\ b'$ such that $b'\{H'\} = b$, by lemma (5.6). Since $\varnothing\ ;\ \varnothing \vdash a : A$ and $\varnothing \vdash \varnothing; \varnothing$ and a is not a value, we have $H, \Gamma, \Gamma_4, Q, a'$ such that

[6] We present the proof for its dependent counterpart later.

$H \vdash \lfloor \Gamma_4 \rfloor ; \Gamma$ and $[\varnothing]\ a \Rightarrow_S [H\,;\,\mathbf{u}\,;\,\Gamma_4]\ a'$ and $\lfloor \Gamma_4 \rfloor\ ;\ \Gamma \vdash a' : A$, by the soundness theorem.

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 $H \vdash \lfloor \Gamma_4 \rfloor ; \Gamma$ and $\lfloor \Gamma_4 \rfloor\ ;\ \Gamma \vdash a' : A$, 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 5.13. *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 $\varnothing \vdash \varnothing ; \varnothing$, either a is a value or there exists H, Γ_4, Q, a' such that $[\varnothing]\ a \Rightarrow_S [H\,;\,\mathbf{u}\,;\,\Gamma_4]\ a'$, in which case, by lemma (5.5), $a \rightsquigarrow a'\{H\}$. □

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

6 APPLICATIONS

6.1 Irrelevance

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 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. For 0 to mean no usage in a semiring Q, the equation $q + 1 = 0$ must have no solution. We call an element $q' \in Q$ *positive* (respectively *positive-or-more*) iff $q' = q + 1$ (respectively $q + 1 \leq q'$) for some $q \in Q$. The above condition then means that 0 is not positive. If we also have a partial order, the constraint $q + 1 \leq 0$ must be unsatisfiable; meaning 0 should not be positive-or-more. We call this the *zero-unusable* criterion. Henceforth, we restrict our attention to semirings that meet this criterion. The following lemmas formalize the idea discussed here.

Lemma 6.1. *In a zero-unusable semiring, if $[H]\ a \Rightarrow_S [H'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ a'$ and $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, then the component $\mathbf{u}'(x_i) = 0$ and $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H'$.*

We see above that locations with count 0 cannot be referenced during computation. Also, the count for such locations always remains 0. Now, if they cannot be referenced, what they contain should not matter. In other words, 0-graded variables do not affect the result of computation. Two initial configurations that differ only in the assignments of some 0-graded variables produce identical results. This means that such assignments do not interfere with evaluation and are irrelevant.

Lemma 6.2 (Zero non-interference). *Let $H_{i1} = x_i \overset{0}{\mapsto} \Gamma_1 \vdash a_1 : A_1$ and $H_{i2} = x_i \overset{0}{\mapsto} \Gamma_2 \vdash a_2 : A_2$. Then, in a zero-unusable semiring, if $[H_1, H_{i1}, H_2]\ b \Rightarrow_{S \cup fv\, a_2} [H_1', H_{i1}, H_2'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ b'$, then $[H_1, H_{i2}, H_2]\ b \Rightarrow_{S \cup fv\, a_1} [H_1', H_{i2}, H_2'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ b'$.*

Note that not just 0-graded resources may be unusable, any s-graded resource for which the constraint $q + 1 \leq s$ is unsatisfiable is unusable. With respect to the security semirings described in Section 3.2, this means that data from any security level s for which $1 \not\leq s$ is unusable. This makes sense since the default view of the type system is 1 or Public so that data judged to be more secure (or incomparable) cannot be used at this level. This gives us the following lemma for the class of security lattices described in Section 3.2:

Lemma 6.3 (s non-interference). *Let $1 \not\leq s$ in a security lattice. Let $H_{i1} = x_i \overset{s}{\mapsto} \Gamma_1 \vdash a_1 : A_1$ and $H_{i2} = x_i \overset{s}{\mapsto} \Gamma_2 \vdash a_2 : A_2$. If $[H_1, H_{i1}, H_2]\ b \Rightarrow_{S \cup fv\, a_2} [H_1', H_{i1}, H_2'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ b'$, then $[H_1, H_{i2}, H_2]\ b \Rightarrow_{S \cup fv\, a_1} [H_1', H_{i2}, H_2'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ b'$.*

6.2 Garbage Collection

Now let us look at locations with count 0 in memory graphs. The sum of the lengths of all paths from the source node to such a node must be 0. The zero-unusable criterion, along with the count balance property, implies that none of these paths has positive-or-more length. This means that all the edge-weights in any such path cannot be positive-or-more.

The condition that 0 is not positive-or-more is a weaker version of a well-known constraint put on semirings. If 0 is a minimal element, a stronger constraint is zerosumfree[7]. A semiring Q is said to be zerosumfree if for any $q_1, q_2 \in Q$, the equation $q_1 + q_2 = 0$ implies $q_1 = q_2 = 0$. If we work with a zerosumfree semiring with 0 as a minimal element, we know that the length of any path from the source node to a node with count 0 is 0. But all the edge-weights along such a path may be non-zero. This is so because the product of two non-zero elements may be 0. If we disallow this, then there is no path from the source node to such a node (0 weight edges are omitted). Semirings which satisfy $q_1 \cdot q_2 = 0 \implies q_1 = 0$ or $q_2 = 0$ are called *entire*[8]. With these constraints on the semiring, we have the following lemma:

Lemma 6.4. *In a zerosumfree, entire semiring with* 0 *as a minimal element, if* $H \vdash \Delta; \Gamma$ *and* $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$*, then* v_i *(the node corresponding to* x_i*) belongs to an isolated subgraph (of* $G_{H,\Gamma}$*) that does not contain the source node.*

The lemma above says that all the 0-count assignments lie in isolated islands disconnected from the line of computation. So at any point, it is safe to garbage collect all such assignments.

6.3 Linearity

Let us now look at linearity. Just having a 1 in the semiring is not enough to capture a notion of linearity. For example, 1 does not really represent linear usage in the boolean semiring since $1 + 1 = 1$. If 1 must mean linear usage, then it cannot be equal to or greater than the successor of any quantity other than 0, where *successor* of q is defined as $q + 1$. Formally, the pair of constraints: $q + 1 \leq 1$ and $q \neq 0$ must have no solution. We call this the *one-linear* criterion. In semirings that meet the zero-unusable and one-linear criteria, 1 represents single usage.

Mirroring our discussion on 0-usage, we strengthen the one-linear criterion to derive a useful property about nodes with a count of 1 in memory graphs. Let us call semirings obeying the following constraints linear:

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

For entire, zerosumfree, linear semirings with 0 and 1 as minimal elements, we have the following property:

Lemma 6.5 (Quantitative single-pointer property). *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.

Now that we have explored a graded simple type system, we move on to dependent types.

[7] Our terminology follows Golan [1999].

[8] The zerosumfree property is sometimes called "positive" and the entire property is sometimes called "zero-product" property.

7 GRADED DEPENDENT TYPES

In this section we define GRAD, a language with graded dependent types. The syntax is presented below. GRAD uses a single syntactic category for terms and types.

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

7.1 Type System

The rules of this type system, shown in Figure 3 on page 20, 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**.[9] We annotate Barendregt's system with quantities, as well as add the unit type, sums and sigma types. Note that the rule T-CONVERT uses the definitional equivalence relation, which is essentially β-equivalence. This relation is axiomatically specified in Section 9.1.

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. 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 must include premises of the form $\Delta\ ;\ \Gamma \vdash A : \mathbf{type}$, that hold when A is a well-formed type. But we don't want to add this usage to the usage of the term.

What this means for the type system is that any usage of a context to check an irrelevant component, like the type, should be multiplied by 0, just like the irrelevant argument in example 4.6. For example, in the rule for variables rule T-VAR, 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 rule T-WEAK, we check that the type of the weakened variable is well-formed using some context Γ_2 that is compatible with Γ_1 (same Δ). Since $\Gamma_1 + 0 \cdot \Gamma_2 = \Gamma_1$, usage Γ_2 doesn't appear in the conclusion of the rule. Many rules follow this pattern of checking types with some usage-unconstrained context, including Γ_2 in rule T-CONVERT and rule T-LAM, and Γ_3 in rule T-UNITELIM. This rule T-UNITELIM implements a form of dependent pattern matching. Here, the type of the branch can observe 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 **Unit** type. The other elimination rules, rule T-SIGMAELIM and rule T-SUMELIM, also follow this style of dependent pattern matching.

Irrelevant Quantification. Now consider the rule T-PI. In particular, note that the usage annotation on the type itself (q) is different from r, which records how many times x is used in B. The annotation q tracks the usage of the argument in the body of a function with this type and this usage may have no relation to the usage of x in the body of the type itself. This difference between q and r allows GRAD to represent parametric polymorphism by marking type arguments with usage 0. For example, the analogue of the System F type $\forall\alpha.\alpha \to \alpha$, can be expressed in this system as $\Pi x{:}^0\, \mathbf{type}.^1 x \to x$. This type is well-formed 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.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

T-TENSOR
$$\frac{\Delta\ ;\ \Gamma_1 \vdash a : A \qquad \Delta\ ;\ \Gamma_2 \vdash b : B\{a/x\} \qquad \Delta, x{:}A\ ;\ \Gamma_3, x{:}^r A \vdash B : \mathbf{type}}{\Delta\ ;\ q \cdot \Gamma_1 + \Gamma_2 \vdash (a, b) : \Sigma x{:}^q A.B}$$

T-SIGMAELIM
$$\frac{\Delta\ ;\ \Gamma_1 \vdash a : \Sigma x{:}^q A_1.A_2 \qquad \Delta, x{:}A_1, y{:}A_2\ ;\ \Gamma_2, x{:}^q A_1, y{:}^1 A_2 \vdash b : B\{(x,y)/z\} \qquad \Delta, z{:}(\Sigma x{:}^q A_1.A_2)\ ;\ \Gamma_3, z{:}^r (\Sigma x{:}^q A_1.A_2) \vdash B : \mathbf{type}}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x,y) = a\ \mathsf{in}\ b : B\{a/z\}}$$

T-SUM
$$\frac{\Delta\ ;\ \Gamma_1 \vdash A_1 : \mathbf{type} \qquad \Delta\ ;\ \Gamma_2 \vdash A_2 : \mathbf{type}}{\Delta\ ;\ \Gamma_1 + \Gamma_2 \vdash A_1 \oplus A_2 : \mathbf{type}}$$

T-INJ1
$$\frac{\Delta\ ;\ \Gamma \vdash a : A_1 \qquad \Delta\ ;\ \Gamma_1 \vdash A_2 : \mathbf{type}}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_1\ a : A_1 \oplus A_2}$$

T-INJ2
$$\frac{\Delta\ ;\ \Gamma \vdash a : A_2 \qquad \Delta\ ;\ \Gamma_1 \vdash A_1 : \mathbf{type}}{\Delta\ ;\ \Gamma \vdash \mathbf{inj}_2\ a : A_1 \oplus A_2}$$

T-SUMELIM
$$\frac{\begin{array}{c}1 \leq q \\ \Delta\ ;\ \Gamma_1 \vdash a : A_1 \oplus A_2 \\ \Delta\ ;\ \Gamma_2 \vdash b_1 : \Pi x{:}^q A_1.B\{\mathbf{inj}_1\ x/y\} \\ \Delta\ ;\ \Gamma_2 \vdash b_2 : \Pi x{:}^q A_2.B\{\mathbf{inj}_2\ x/y\} \\ \Delta, y{:}A_1 \oplus A_2\ ;\ \Gamma_3, y{:}^r A_1 \oplus A_2 \vdash B : \mathbf{type}\end{array}}{\Delta;q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B\{a/y\}}$$

Fig. 3. Typing rules for dependent, quantitative type system

Some versions of irrelevant quantifiers in type theories constrain r to be equal to q [Abel and Scherer 2012]. By coupling the usage of variables in the body of the lambda with the result type of the Π, these systems rule out the representation of polymorphic types, such as the one shown above. Here, 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 $\Pi x :^1 (\Box^0 A).B$ (and a similar distinction between $\Sigma x{:}^0 A.B$ and $\Sigma x{:}^1(\Box^0 A).B$) that motivates our inclusion of the usage annotation on the Π and Σ types directly. In the simple type system, we can derive usage-annotated functions from linear functions and the box modality: there is no need to annotate the arrow with any quantity other than 1. But here, due to dependency, we cannot have parametrically polymorphic types without this additional form. On the other hand, with the presence of usage-annotated Σ-types, we do not need to include the box modality. Instead, we can encode the type $\Box^q A$ using the non-dependent tensor $\Sigma x{:}^q A.\mathbf{Unit}$. Thus, we can eliminate this special form from the language.

7.2 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 7.1 (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 7.2 (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 anywhere in the judgement.

Lemma 7.3 (Weakening). *If* $\Delta_1, \Delta_2\ ;\ \Gamma_1, \Gamma_2 \vdash a : A$ *and* $\Delta_1\ ;\ \Gamma_3 \vdash B : \mathbf{type}$ *then* $\Delta_1, x{:}^0 B, \Delta_2\ ;\ \Gamma_1, x :^0 B, \Gamma_2 \vdash a : A$.

With a small-step relation that is identical to that of the simply typed version, we have the following type soundness theorem.

Theorem 7.4 (Preservation). *If* $\Delta\ ;\ \Gamma \vdash a : A$ *and* $a \rightsquigarrow a'$ *then* $\Delta\ ;\ \Gamma \vdash a' : A$.

Theorem 7.5 (Progress). *If* $\varnothing\ ;\ \varnothing \vdash a : A$ *then either* a *is a value or there exists some* a' *such that* $a \rightsquigarrow a'$.

Now, akin to the simple version, we develop a heap semantics for the dependent version.

8 HEAP SEMANTICS FOR GRAD

The presence of dependent types causes one issue with the heap semantics: because substitutions are delayed through the heap, the terms and their types can "get out of sync".

For example, if we have the application of a polymorphic identity function $\lambda x :^0 \mathbf{type}.\lambda y :^1 x.y$ to some type argument $\mathbf{Unit}$, then the result $(\lambda x :^0 \mathbf{type}.\lambda y :^1 x.y)\ \mathbf{Unit}$ should have type $\Pi y :^1 \mathbf{Unit}.\mathbf{Unit}$. By the rule Small-AppBeta, $[\varnothing]\ (\lambda x :^0 \mathbf{type}.\lambda y :^1 x.y)\ \mathbf{Unit} \Rightarrow [x \overset{0}{\mapsto} \mathbf{Unit}]\ \lambda y :^1 x.y$. The term $\lambda y :^1 x.y$ has type $\Pi y :^1 x.x$. But since $x = \mathbf{Unit}$, we see that $\Pi y :^1 x.x = \Pi y :^1 \mathbf{Unit}.\mathbf{Unit}$; as such,

$\lambda y :^1 x.y$ can also be assigned the type $\Pi y :^1 \textbf{Unit}.\textbf{Unit}$. So to align the types of the redex and the reduct, we need to know about the new assignments loaded into the heap. This issue did not exist in the simple setting since the types did not depend on term variables. Note that this is not a usage-related issue, any heap-based reduction that delays substitution will need to address it while proving soundness. The good news is that it can be resolved with a simple extension to the type system.

8.1 A Dependently-Typed Language with Definitions

We extend our contexts with definitions that mimic delayed substitutions. These definitions are used *only* in deriving type equalities. From the type system perspective, they are essentially a bookkeeping device added to enable reasoning with respect to the heap semantics.

$$\begin{array}{llll} \textit{usage contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x :^q A \mid \Gamma, x = a :^q A \\ \textit{contexts} & \Delta & ::= & \varnothing \mid \Delta, x{:}A \mid \Delta, x = a : A \end{array}$$

Along with this extension to the context, we modify the conversion rule and add two new typing rules to the system, as shown below. (In rule T-conv, $A\{\Delta\}$ denotes the type obtained by substituting in A, in reverse order, the definiens in place of the variables for the definitions in Δ.)

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

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

$$\textsc{T-def}\ \frac{x \notin \operatorname{dom}\Delta \qquad \Delta \,;\, \Gamma \vdash a : A}{\Delta, x = a : A \,;\, 0 \cdot \Gamma, x = a :^1 A \vdash x : A}$$

$$\textsc{T-weak-def}\ \frac{x \notin \operatorname{dom}\Delta \qquad \Delta \,;\, \Gamma_1 \vdash b : B \qquad \Delta \,;\, \Gamma_2 \vdash a : A}{\Delta, x = a : A \,;\, \Gamma_1, x = a :^0 A \vdash b : B}$$

The definitions act like usual variable assumptions: rule T-def and rule T-weak-def mirror rule T-var and rule T-weak respectively. They are applied only during the conversion rule T-conv that substitutes out these definitions before comparing for β-equivalence. This modified rule means that the term $\lambda y :^1 x.y$ can be given the type $\Pi y :^1 \textbf{Unit}.\textbf{Unit}$ in a context that defines x to be **Unit**.

The extended type system too has the syntactic soundness properties mentioned in Section 7.2. Furthermore, because definitions act only on types, definitions do not add extra resource demands to the typing derivation. As a result, we can always convert a normal variable assumption to include some definition as long as the definiens type checks. Furthermore, the resources used by the definiens (Γ below) are unimportant.

Lemma 8.1 (InsertEq). *If* $\Delta_1, x{:}A, \Delta_2 \,;\, \Gamma_1, x :^q A, \Gamma_2 \vdash b : B$ *and* $\Delta_1 \,;\, \Gamma \vdash a : A$, *then* $\Delta_1, x = a : A, \Delta_2 \,;\, \Gamma_1, x = a :^q A, \Gamma_2 \vdash b : B$.

Contexts can also be weakened with new (unused) definitions, analogous to lemma 7.3.

Lemma 8.2 (Weakening with Definitions). *If* $\Delta_1, \Delta_2 \,;\, \Gamma_1, \Gamma_2 \vdash b : B$ *and* $\Delta_1 \,;\, \Gamma \vdash a : A$ *then* $\Delta_1, x = a : A, \Delta_2 \,;\, \Gamma_1, x = a :^0 A, \Gamma_2 \vdash b : B$.

Because we have modified the contexts to include definitions, we need to modify the heap reduction and compatibility relations. Only for the reduction rules that load new assignments into the heap, we now need the added context of new variables (Γ_4) to remember their assignments. For example, the rule Small-AppBeta is modified as below.

$\boxed{[H]\ a \Rightarrow_S^q [H'\,;\,\mathbf{u}'\,;\,\Gamma_4]\ a'}$ *(SmallStep with definitions)*

Small-DAppBeta

$$\frac{\begin{array}{c} x \notin \mathsf{Var}\, H \cup \mathrm{fv}\, b \cup \mathrm{fv}\, a - \{y\} \cup S \\ a' = a\{x/y\} \end{array}}{[H]\ (\lambda y{:}^q A_1.a)\ b \Rightarrow_S^r [H, x \overset{r\cdot q}{\mapsto} \Gamma \vdash b : A\,;\, \mathbf{0}^{|H|} \diamond 0\,;\, x{=}b{:}^{r\cdot q} A]\ a'}$$

Similarly, rule Compat-Cons needs to track more information in the context.

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

Compat-ConsDef

$$\frac{\begin{array}{c} H \vdash \Delta; \Gamma_1 + (q \cdot \Gamma_2) \\ \Delta\ ;\ \Gamma_2 \vdash a : A \\ x \notin \mathrm{dom}\, H \end{array}}{H, x \overset{q}{\mapsto} \Gamma_2 \vdash a : A \vdash \Delta, x{=}a{:}A; \Gamma_1, x{=}a{:}^q A}$$

Note that with this modification, if $H \vdash \Delta; \Gamma$ then $b\{H\} = b\{\Delta\}$ for any term b.

These are all the changes we need. Since the added context of new variables does not play a major role in the step relation, all the previously stated lemmas regarding this relation hold. But with dependency, the multi-substitution lemma 5.8 needs to be modified to also substitute into the type (in addition to the term).

Lemma 8.3 (Multi-substitution). *If $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$, then $\varnothing\ ;\ \varnothing \vdash a\{H\} : A\{H\}$.*

Another point worth noting here is that, if $[H]\ a \Rightarrow_S^q [H'\,;\,\mathbf{u}'\,;\,\Gamma']\ a'$ then $a\{H\} \equiv a'\{H'\}$ by lemma 5.5 and definition 9.1. Before moving further, let us reflect how the original typing and heap compatibility judgements relate to their extended counterparts. For the sake of distinction, let us denote the original relations by $\vdash_o$. Now, for $H \vdash_o \Delta; \Gamma$, let Δ_H and Γ_H be Δ and Γ respectively with their variables defined according to (assignments in) H. Also, let H_H denote H with the variables in the embedded contexts in H defined according to H. Then, we have:

Lemma 8.4 (Elaboration). *If $H \vdash_o \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash_o a : A$, then $H_H \vdash \Delta_H; \Gamma_H$ and $\Delta_H\ ;\ \Gamma_H \vdash a : A$.*

By virtue of this elaboration, soundness for the extended system implies soundness for the original one.

8.2 Proof of the Heap Soundness Theorem

Now we prove the heap soundness theorem for GrAD. However, we first state some subordinate lemmas that are required in the proof. The following lemma allows us to throw away resources from heaps.

Lemma 8.5 (Sub-heaping). *If $H \vdash \Delta; \Gamma$ and $\Gamma' \leq \Gamma$, then there exists H' such that $H' \vdash \Delta; \Gamma'$ and $H' \leq H$.*

We can insert new definitions into the heap.

Lemma 8.6 (SmallStep weakening). *If $[H_1, H_2]\ a \Rightarrow_{S \cup \{x\}}^r [H_1', H'\,;\,\mathbf{u}_1 \diamond \mathbf{u}\,;\,\Gamma_4]\ a'$ and $|H_1'| = |\mathbf{u}_1| = |H_1|$ and $x \notin \mathrm{dom}\, H_1, H_2$ and $fv\, a_1 \cap \mathrm{dom}\, H_2 = \emptyset$, then*

$$[H_1, x \overset{q}{\mapsto} \Gamma_1 \vdash a_1 : A_1, H_2]\ a \Rightarrow_S^r [H_1', x \overset{q}{\mapsto} \Gamma_1 \vdash a_1 : A_1, H'\,;\,\mathbf{u}_1' \diamond 0 \diamond \mathbf{u}\,;\,\Gamma_4]\ a'$$

Lemma 8.7 (Compatibility weakening). *Let* $H_1, H_2 \vdash \Delta, \Delta'; ((r \cdot \Gamma_{11}) + \Gamma_0), \Gamma_1$ *and* $\Delta\ ;\ \Gamma_{11} \vdash a : A$ *and* $|H_1| = |\Gamma_0| = |\Delta|$ *and* $x \notin \operatorname{dom} H_1, H_2$. *Let* H_2' *be* H_2 *with the embedded contexts weakened by inserting* $x = a :^0 A$ *at the* $|\Delta|$ *position. Then,*

$$H_1, x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A, H_2' \vdash \Delta, x = a : A, \Delta'; \Gamma_0, x = a :^r A, \Gamma_1$$

The soundness theorem follows as an instance of the invariance lemma below. The lemma provides a strong enough hypothesis for the induction to go through.

Lemma 8.8 (Invariance). *If* $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma$ *and* $\Delta\ ;\ \Gamma \vdash a : A$ *and* $1 \le q$ *and* $\operatorname{dom} \Delta \subseteq S$, *then either* a *is a value or there exists* Γ', H', $\mathbf{u}'$, Γ_4 *and* a' *such that:*

- $[H]\ a \Rightarrow_S^q [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ a'$
- $H' \vdash \Delta, \lfloor \Gamma_4 \rfloor; (\Gamma_0, 0 \cdot \Gamma_4) + q \cdot \Gamma'$
- $\Delta, \lfloor \Gamma_4 \rfloor\ ;\ \Gamma' \vdash a' : A$
- $q \cdot \overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_4} \times \langle H' \rangle \le q \cdot (\overline{\Gamma} \diamond \mathbf{0}) + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_4}$
- $\operatorname{dom} \Gamma_4$ *is disjoint from* S

Proof. Let $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$. We prove this lemma by induction on the typing judgement $\Delta\ ;\ \Gamma \vdash a : A$. For brevity, we show only the most interesting cases.

- **rule** T-sub
  Let $\Delta\ ;\ \Gamma_2 \vdash a : A$ where $\Delta\ ;\ \Gamma_1 \vdash a : A$ and $\Gamma_1 \le \Gamma_2$. Further, $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma_2$.
  Since $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma_2$ and $\Gamma_1 \le \Gamma_2$, by lemma 8.5, there exists H' such that $H' \vdash \Delta; \Gamma_0 + q \cdot \Gamma_1$ and $H' \le H$. By inductive hypothesis, $[H']\ a \Rightarrow_S^q [H''\ ;\ \mathbf{u}\ ;\ \Gamma_4]\ a''$. Since $H' \le H$, we have, $[H]\ a \Rightarrow_S^q [H''\ ;\ \mathbf{u}\ ;\ \Gamma_4]\ a''$. The remaining clauses follow from the inductive hypothesis and the fact that $\Gamma_1 \le \Gamma_2$.
- We don't need to consider rule T-var and rule T-weak since for $H \vdash \Delta; \Gamma$, any variable $x \in \operatorname{dom} \Delta$ should be a definition of the form $x = a : A$.
- **rule** T-def
  Let $\Delta, x = a : A\ ;\ 0 \cdot \Gamma, x = a :^1 A \vdash x : A$ where $\Delta\ ;\ \Gamma \vdash a : A$ and $x \notin \operatorname{dom} \Delta$. Further, $H \vdash \Delta, x = a : A; \Gamma_0 + q \cdot (0 \cdot \Gamma, x = a :^1 A)$. Let $\Gamma_0 = \Gamma_0', x = a :^r A$. So $H \vdash \Delta, x = a : A; \Gamma_0', x = a :^{(r+q)} A$.
  Therefore, $H = H_1, x \overset{(r+q)}{\mapsto} \Gamma_{11} \vdash a : A$ where $\Delta\ ;\ \Gamma_{11} \vdash a : A$.
  Since $1 \le q$, we have, $[H_1, x \overset{(r+q)}{\mapsto} \Gamma_{11} \vdash a : A]\ x \Rightarrow_S^q [H_1, x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A\ ;\ \mathbf{0}^{|H_1|} \diamond q\ ;\ \varnothing]\ a$.
  Also, $H_1, x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A \vdash \Delta, x = a : A; (\Gamma_0', x = a :^r A) + q \cdot (\Gamma_{11}, x = a :^0 A)$. By weakening, we have, $\Delta, x = a : A\ ;\ \Gamma_{11}, x = a :^0 A \vdash a : A$. The fourth clause: $q \cdot (\overline{\Gamma_{11}} \diamond 0) + (\mathbf{0} \diamond q) \le \mathbf{0} \diamond q + (\mathbf{0} \diamond q) \times \begin{pmatrix} \langle H_1 \rangle & \mathbf{0}^\intercal \\ \overline{\Gamma_{11}} & 0 \end{pmatrix}$ follows by reflexivity.
- **rule** T-weak-def
  Let $\Delta, x = a : A\ ;\ \Gamma, x = a :^0 A \vdash b : B$ where $\Delta\ ;\ \Gamma \vdash b : B$ and $\Delta\ ;\ \Gamma_9 \vdash a : A$ and $x \notin \operatorname{dom} \Delta$. Further, $H \vdash \Delta, x = a : A; \Gamma_0 + q \cdot (\Gamma, x = a :^0 A)$. Let $\Gamma_0 = \Gamma_0', x = a :^r A$. So $H \vdash \Delta, x = a : A; \Gamma_0' + q \cdot \Gamma, x = a :^r A$. Therefore, $H = H_1, x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A$ where $\Delta\ ;\ \Gamma_{11} \vdash a : A$. Also, $H_1 \vdash \Delta; \Gamma_0' + q \cdot \Gamma + r \cdot \Gamma_{11}$.
  Applying the inductive hypothesis, we get, $[H_1]\ b \Rightarrow_{S \cup \{x\}}^q [H_1', H_4\ ;\ \mathbf{u}_1 \diamond \mathbf{u}_4\ ;\ \Gamma_4]\ b'$ and $H_1', H_4 \vdash \Delta, \lfloor \Gamma_4 \rfloor; (\Gamma_0' + r \cdot \Gamma_{11}, 0 \cdot \Gamma_4) + q \cdot (\Gamma', \Gamma'')$ and $\Delta, \lfloor \Gamma_4 \rfloor\ ;\ \Gamma', \Gamma''\ \vdash b' : B$. Here, $|H_1'| = |\mathbf{u}_1| = |H_1| = |\Gamma'| = |\Delta|$. Now, note that x does not appear in H_4. Let H_4' be H_4 with the embedded contexts weakened by inserting $x = a :^0 A$ at the $|\Delta|$ position. Then, by 8.7, $H_1', x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A, H_4' \vdash \Delta, x = a : A, \lfloor \Gamma_4 \rfloor; (\Gamma_0', x = a :^r A, 0 \cdot \Gamma_4) + q \cdot (\Gamma', x = a :^0 A, \Gamma'')$. Because extra assignments do not impact the evaluation, we have, by 8.6,

$[H_1, x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A]\ b \Rightarrow_S^q [H_1', x \overset{r}{\mapsto} \Gamma_{11} \vdash a : A, H_4'\ ;\ \mathbf{u}_1 \diamond 0 \diamond \mathbf{u}_4\ ;\ \Gamma_4]\ b'$. Also, by weakening lemma 8.2, $\Delta, x{=}a{:}A, \lfloor\Gamma_4\rfloor\ ;\ \Gamma', x{=}a{:}^0 A, \Gamma'' \vdash b' : B$. The fourth clause follows from the inductive hypothesis after inserting 0 at the $|\Delta|$-position on both sides.

- **rule** T-APP
  Let $\Delta\ ;\ \Gamma_1 + r \cdot \Gamma_2 \vdash b\,a : B\{a/x\}$ where $\Delta\ ;\ \Gamma_1 \vdash b : \Pi x :^r A.B$ and $\Delta\ ;\ \Gamma_2 \vdash a : A$. Further, $H \vdash \Delta; \Gamma_0 + q \cdot (\Gamma_1 + r \cdot \Gamma_2)$. Now, there are two cases to consider depending on whether b is a value or not.
  - b is not a value.
    In this case, we get from the inductive hypothesis, $[H]\ b \Rightarrow^q_{S \cup \mathrm{fv}\, a} [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ b'$ and $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma_1' \vdash b' : \Pi x :^r A.B$ and $H' \vdash \Delta, \lfloor\Gamma_4\rfloor; (\Gamma_0 + (q \cdot r) \cdot \Gamma_2, 0 \cdot \Gamma_4) + q \cdot \Gamma_1'$. So $[H]\ b\,a \Rightarrow^q_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ b'\,a$ By weakening, we get, $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma_2, 0 \cdot \Gamma_4 \vdash a : A$. Therefore, by App, we have, $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma_1' + r \cdot (\Gamma_2, 0 \cdot \Gamma_4) \vdash b'\,a : B\{a/x\}$. Also, by rearranging, we get $H' \vdash \Delta, \lfloor\Gamma_4\rfloor; (\Gamma_0, 0 \cdot \Gamma_4) + q \cdot (\Gamma_1' + r \cdot (\Gamma_2, 0 \cdot \Gamma_4))$. The fourth clause follows from the corresponding clause of inductive hypothesis.
  - b is a value.
    Since b has a Π-type, it must be headed by a λ. Let $b = \lambda y :^s A_1.b_1$ for some sufficiently fresh variable y. Then, we have, $\Delta\ ;\ \Gamma_1 \vdash \lambda y :^s A_1.b_1 : \Pi x :^r A.B$. By inversion, there exists B_1 such that $\Delta, y{:}A_1\ ;\ \Gamma_1, y :^s A_1 \vdash b_1 : B_1$ and $\Delta\ ;\ \Gamma_7 \vdash \Pi y :^s A_1.B_1 :$ **type** and $(\Pi y :^s A_1.B_1)\{\Delta\} \equiv (\Pi x :^r A.B)\{\Delta\}$. By definition 9.1, $s = r$ and $A_1\{\Delta\} \equiv A\{\Delta\}$ and $B_1\{\Delta\} \equiv B\{y/x\}\{\Delta\}$. Now, by rule T-CONV, $\Delta\ ;\ \Gamma_2 \vdash a : A_1$. Therefore, by lemma 8.1, $\Delta, y{=}a{:}A_1\ ;\ \Gamma_1, y{=}a{:}^r A_1 \vdash b_1 : B_1$.
    We have, $[H]\ (\lambda y :^r A_1.b_1)\ a \Rightarrow^q_S [H, y \overset{(q\cdot r)}{\mapsto} \Gamma_2 \vdash a : A_1\ ;\ \mathbf{0}\ ;\ y{=}a{:}^{(q\cdot r)} A_1]\ b_1$. Again, since $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma_1 + (q \cdot r) \cdot \Gamma_2$, we get, $H, y \overset{(q\cdot r)}{\mapsto} \Gamma_2 \vdash a : A_1 \vdash \Delta, y = a : A_1; (\Gamma_0, y = a :^0 A_1) + q \cdot (\Gamma_1, y{=}a{:}^r A_1)$.
    By regularity, we know that $\Delta\ ;\ \Gamma' \vdash B\{a/x\}$: **type**. By weakening, $\Delta, y{=}a{:}A_1\ ;\ \Gamma', y{=}a{:}^0 A_1 \vdash B\{a/x\}$: **type**. But $B\{a/x\}\{\Delta, y = a : A_1\} = B\{a/x\}\{\Delta\} = B\{y/x\}\{a/y\}\{\Delta\} = B\{y/x\}\{\Delta\}\{a\{\Delta\}/y\} \equiv B_1\{\Delta\}\{a\{\Delta\}/y\} = B_1\{a/y\}\{\Delta\} = B_1\{\Delta, y = a : A_1\}$. Hence, by rule T-CONV, $\Delta, y{=}a{:}A_1\ ;\ \Gamma_1, y{=}a{:}^r A_1 \vdash b_1 : B\{a/x\}$. The fourth clause: $q \cdot (\overline{\Gamma_1} \diamond r) + \mathbf{0} + (\mathbf{0} \diamond (q \cdot r)) \times \left(\begin{smallmatrix} \langle H\rangle & \mathbf{0}^\intercal \\ \overline{\Gamma_2} & 0 \end{smallmatrix}\right) \leq q \cdot ((\overline{\Gamma_1} + r \cdot \overline{\Gamma_2}) \diamond 0) + \mathbf{0} + (\mathbf{0} \diamond (q \cdot r))$ follows by reflexivity.
- **rule** T-CONV
  Let $\Delta\ ;\ \Gamma \vdash a : B$ where $\Delta\ ;\ \Gamma \vdash a : A$ and $\Delta\ ;\ \Gamma_1 \vdash B :$ **type** and $A\{\Delta\} \equiv B\{\Delta\}$. Further, $H \vdash \Delta; \Gamma_0 + q \cdot \Gamma$. Therefore, by inductive hypothesis, $[H]\ a \Rightarrow^q_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ a'$ and $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma' \vdash a' : A$ and $H' \vdash \Delta, \lfloor\Gamma_4\rfloor; (\Gamma_0, 0 \cdot \Gamma_4) + q \cdot \Gamma'$.
  Now, since $\mathrm{fv}\, A \subseteq \mathrm{dom}\, \Delta$ and $\mathrm{fv}\, B \subseteq \mathrm{dom}\, \Delta$; we have, $A\{\Delta, \lfloor\Gamma_4\rfloor\} = A\{\Delta\}$ and $B\{\Delta, \lfloor\Gamma_4\rfloor\} = B\{\Delta\}$. Therefore, $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma' \vdash a' : B$. The other clauses follow from the inductive hypothesis.

□

THEOREM 8.9 (SOUNDNESS). *If $H \vdash \Delta; \Gamma$ and $\Delta\ ;\ \Gamma \vdash a : A$ and $S \supseteq \mathrm{dom}\, \Delta$, then either a is a value or there exists Γ', H', $\mathbf{u}'$, Γ_4, A' such that:*

- $[H]\ a{:}\ A \Rightarrow_S [H'\ ;\ \mathbf{u}'\ ;\ \Gamma_4]\ a'{:}\ A'$
- $H' \vdash \Delta, \lfloor\Gamma_4\rfloor; \Gamma'$
- $\Delta, \lfloor\Gamma_4\rfloor\ ;\ \Gamma' \vdash a' : A'$
- $\overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_4} \times \langle H'\rangle \leq \overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H'\rangle + \mathbf{0} \diamond \overline{\Gamma_4}$

PROOF. Follows from 8.8 with $q = 1$ and $\Gamma_0 = 0 \cdot \Gamma$. □

The soundness theorem is similar in spirit to theorems showing correctness of usage in graded type systems via operational methods. But this theorem can be proved by simple induction on the typing derivation; it does not require much extra machinery over and above the reduction relation, unlike the proof of soundness in Brunel et al. [2014] which requires a realizability model on top of the reduction relation. In this regard, our soundness theorem is more in line with the modality preservation theorem in Abel and Bernardy [2020].

We can use the soundness theorem to prove the usual preservation and progress lemmas. The proofs are similar to the corresponding ones for the simple version (5.12 and 5.13). This means that the ordinary semantics is sound with respect to a resource-aware semantics.

Below, we show an example application of this theorem. Other similar examples can be worked out using the lemmas from Section 6.

Example 8.10. Consider any security lattice Q, as described in Section 3.2. Let s be any element such that $1 \not\leq s$. Also, let $\varnothing \vdash A : {}^1\mathbf{type} \to \mathbf{type}$ such that $A\,\mathbf{Unit} = \mathbf{Int}$. Now, let $x{:}^1\,\mathbf{type}$, $y{:}^q A\,x \vdash B : \mathbf{type}$ for some $q \in Q$.

In empty context, consider a term f of type $\Pi x{:}^r \mathbf{type}.\Pi y{:}^s A\,x.B$ for some $r \in Q$. (Note that neither r and 1 nor s and q need to be equal, as explained in irrelevant quantification.) Then, we can show that both $f\,\mathbf{Unit}\,0$ and $f\,\mathbf{Unit}\,1$ either diverge or produce equal values. If $f\,\mathbf{Unit}$ diverges, then both diverge. Otherwise, $[\varnothing]\,f\,\mathbf{Unit} \Longrightarrow [H]\,\lambda y :^s C.b$ where $C\{H\} \equiv \mathbf{Int}$. Now, $[H]\,(\lambda y{:}^s C.b)\,0 \Rightarrow [H, y \overset{s}{\mapsto} 0]\,b$ and $[H]\,(\lambda y{:}^s C.b)\,1 \Rightarrow [H, y \overset{s}{\mapsto} 1]\,b$. By 6.3, we know that both $((H, y \overset{s}{\mapsto} 0), b)$ and $((H, y \overset{s}{\mapsto} 1), b)$ either diverge or reduce to the same value.

9 DISCUSSION

9.1 Definitional-Equivalence and Irrelevance

The terms "irrelevance" and "irrelevant quantification" have multiple meanings in the literature. Our primary focus is on erasability, the ability 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 9.1. 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 $\Sigma x{:}^r A.B$ 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 6 tell us that such coarsening of the equivalence relation is consistent with evaluation, and therefore contains the step relation.

9.2 Connection to Haskell

The current design of linear types in GHC/Haskell is essentially an instance of the type system described in this paper, one that uses the linearity semiring. Haskell 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 with dependent types that we need a uniform treatment which we can achieve through this graded type system. The current Haskell structure will be able to migrate to a graded type system 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 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 mentioned trouble with static compilation. Furthermore, we see no difficulty in extending our graded type system 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.

Commentary on the practicalities of type checking Haskell based on GRAD appears in the extended version of this paper.

9.3 Comparison with Quantitative Type Theory

Quantitative Type Theory QTT [Atkey 2018; McBride 2016] uses elements of a resource semiring to track the usage of variables in a dependent type system. This system has a typing judgement of the form: $x_1 :^{\rho_1} A_1, x_2 :^{\rho_2} A_2, \ldots, x_n :^{\rho_n} A_n \vdash a :^{\sigma} A$, where ρ_is and σ 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 σ copies of a of type A.

In QTT, σ can be either 0 or 1. When σ is 1, the system is similar to GRAD. GRAD does not have the 0-fragment of QTT but this is not a limitation per se: to express the requirement of 0 copy of a, one need only multiply the context by 0. This approach implies that our system treats types the same as any other irrelevant component of terms.

In contrast, QTT disables resource checking for the 0-fragment. In other words, in the 0-fragment, the resource annotations are not meaningful. This difference has both positive and negative effects on the design of the language.

On the positive side, because linear tensor types are turned into normal (non-linear) products, QTT can support *strong*-Σ types, allowing projections that violate the usage requirements of

their construction. In contrast, GrAD supports *weak*-Σ types only, with resource-checked pattern matching as the only elimination form.

On the negative side, however, QTT is restricted to semirings that are *zerosumfree* ($q_1 + q_2 = 0 \implies q_1 = q_2 = 0$) and *entire* ($q_1 \cdot q_2 = 0 \implies q_1 = 0 \vee q_2 = 0$). (These properties are necessary to prove substitution.) This limits QTT's applicability. For example, QTT can not be applied to the class of semirings described in Section 3.2 that are not entire. On the other hand, our soundness theorem places no constraint on the semiring allowing us to work with such semirings, as lemma 6.3 and example 8.10 show.

Furthermore, because QTT ignores usages in the 0-fragment, its internal logic is limited in its reasoning about the resource usage of programs. For example, the following proposition is not provable in QTT:

$$\forall f : {}^0\mathbf{Bool} \to \mathbf{Bool}.f\,\mathsf{True} = f\,\mathsf{False}$$

The above proposition says that for any constant boolean function, the result of applying it to True is the same as the result of applying it to False. This proposition is not provable in QTT because f ranges over many functions, including those that examine the argument. In the 0-fragment, the type system cannot prevent a function that uses its argument to be given a type that says that it does not.

$$\vdash \lambda x :^0 A.x \;:^0\; \Pi x :^0 A.A$$

Abel [2018] also lists additional ramifications of eliminating resource checking in types. In particular, he notes that in QTT, it is not possible to use resource usage to optimize the computation of types during type checking. In particular, erasing irrelevant terms not only optimizes the output of a compiler for a dependently-typed language, it is also an optimization that is useful during compilation, when types are normalized for comparison.

Finally, we note that GrAD includes case expressions and sub-usaging, while QTT does not.

9.4 Abstract Algebraic Generalization

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

Our contexts are an example of a general algebraic structure, called a partially-ordered left semimodule. Additionally, 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 9.2 (Left Q-semimodule). Given a semiring $(Q, +, \cdot, 0, 1)$, 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 $m \in M$, we have, $0 \odot m = q \odot \bar{0} = \bar{0}$.

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

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 9.3 (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 $\_\alpha : M \longrightarrow N$ 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 natural numbers i, j, k and Q-homomorphisms $\_\alpha : Q^i \longrightarrow Q^j$ and $\_\beta : Q^j \longrightarrow Q^k$, the composition $\_(\alpha \circ \beta) : Q^i \longrightarrow Q^k$ 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 $\mathrm{Hom}(Q^m, Q^n) = \mathcal{M}_{m,n}(Q)$, the set of $m \times n$ matrices over Q, a category. We worked in this category. There may be other such categories worth exploring.

10 OTHER RELATED WORK

10.1 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 a graded type system by allowing resources to be drawn from an arbitrary semiring. We derive a quantitative version of the single pointer property in Section 6. 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.

10.2 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. Gaboardi et al. [2013] 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. But all of these work had some separation between the linear and non-linear parts of their languages. Quantitative type theory [Atkey 2018; McBride 2016] provided a fresh way to look at this problem by combining the linear and non-linear parts using a resource semiring.

10.3 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. Implicit arguments are those that do not appear free in the body of their abstractions. In Miquel's system, only the relevant parts of the computation may be explicit in terms, everything else must be implicit. 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. We have followed their design in making the usage of irrelevant variables in the co-domain of Π-types unrestricted. Specifically, the irrelevance (−) tag in their language corresponds to the 0 grade in our language.

11 FUTURE WORK AND CONCLUSIONS

Graded type systems are a generic framework for expressing the flow and usage of resources in programs. This work provides a new way of incorporating this framework into dependently-typed languages, with the goal of supporting both erasure and linearity in the same system.

We designed a graded dependent type system GRAD and presented a standard substitution-based semantics and a usage-aware heap-based semantics. The standard semantics does not have the ability to model use of resources. But the heap-based semantics can track usage during evaluation of terms. Further, the heap-based reduction relation enforces fair usage of resources. We show that the type system is sound with respect to this heap semantics. This implies that the type system does a proper static accounting of resource usage.

As always, there is more to explore: What additional reasoning principles can we get from our heap semantics? What happens when we add imperative features—like arrays—to our language? What would a general form of equality up to erasure look like? What happens when we add multiple modalities, all of them graded, to our language?

The answers to these questions may have theoretical as well as practical implications. Currently, languages such as Haskell, Rust, Idris, and Agda are experimenting with dependent and linear types, as well as the more general applications of graded type theories. We hope that this work will provide guidance in these language designs and extensions.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1521539, and Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

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* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel. 2018. Resourceful Dependent Types. Presentation at 4th International Conference on Types for Proofs and Programs (TYPES 2018), Braga, Portugal..

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

The Agda-Team. 2020. *Run-time Irrelevance.* https://agda.readthedocs.io/en/v2.6.1.1/language/runtime-irrelevance.html

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

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. 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

Conor McBride. 2016. *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).

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

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

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* (Baltimore, Maryland, USA) *(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:2005.02247 [cs.PL]