Towards Graded Modal Dependent Types

Extended Abstract

Benjamin Moon
University of Kent

Harley Eades III
Augusta University

Dominic Orchard
University of Kent

1 Introduction

In most programming languages, data can flow arbitrarily to any part of a program, being copied and discarded at will. It has been long observed that this 'free flow' of data is a source of program error, as some data has additional constraints. *Linear types* [15, 25] recognise that some data should not be allowed to flow arbitrarily, but should instead be restricted to *linear flow*, being consumed once and never copied or discarded. *Bounded Linear Logic* (BLL) provides more flexibility, tracking the maximum number of allowed uses for a piece of data [16]. BLL can be generalised to allow semiring-based analyses of the flow of data through a program, referred to as *coeffect analyses* [6, 7, 14, 22, 24], which include reuse, information flow security [22], hardware scheduling [14], and sensitivity in differential privacy [10, 11]. Such systems track the dependency of runtime computations on data, but not the dependency of types on data.

Linear data-flow is rare in dependently-typed settings: e.g., the body of the polymorphic identity function in a Martin-Löf style theory $a : \mathsf{Type}, x : a \vdash x : a$ uses a twice (typing x in the context and the subject of the judgment), and x linearly in the subject but not at all in the type. There have been various attempts to reconcile linear and dependent types [8, 9, 18, 19], usually keeping linear and dependent types separate, allowing types to depend only on non-linear variables. These theories are unable to distinguish variables used for computation and those used just for type formation, which could then be erased at runtime.

Recent work by McBride [20], refined by Atkey [2], generalises coeffect analyses to a dependently-typed setting. This approach, called Quantitative Type Theory (QTT), types the above as $a \overset{0}{:} \mathsf{Type}, x \overset{1}{:} a \vdash x \overset{1}{:} a$. The annotation 0 on a explains that we can use a to form a type, but we cannot, or do not, use it at the term level, thus it can be erased at runtime. The cornerstone of QTT's approach is that data-flow of a term to the type level counts as 0 use, so arbitrary type-level use is allowed whilst still enabling quantitative analysis of term-level data-flow.

Abel [1] follows on from the work on QTT, noting some of its shortcomings in describing all type-level usage with 0, including how we lose out on the reasoning benefits of linear and quantitative typing when writing types and type-checking code. For example, optimisations aided by linear types (e.g., static allocation and allocation reuse [17], and erasure [5]) are no longer available at the type level, requiring further analysis. Abel introduces a dependent type theory where terms and types are indexed by resource contexts, which capture variable use in every type and term.

Our approach, called *Graded Modal Dependent Type Theory* (GRTT), has similarities to the work of Abel outlined above—we also decouple resource information from contexts, but rather than annotating terms and types, we annotate contexts *globally*, describing variable use in the context, term, and type of a judgment. This pervasive quantitative tracking enables fine-grained quantitative analysis in both the computational *and* type levels, meaning type-level use is not just collapsed to 0 as in QTT. In addition to resource annotations, we provide rules for *graded modalities*, allowing the inspection of subparts of a term, not just terms as a whole; we use the terminology of "grading", a notion of augmenting types with additional information to capture the structure of proofs or terms [12, 22]. Combining linear, graded, and dependent types and graded modalities provides a powerful substrate for specifying and reasoning about program behaviour. For example, it goes towards enabling linearity-based optimisations to speed up type-checking, inference, and synthesis; support for type case without loss of parametricity; and fine-grained resource policies on types which can be specialised to different domains and type theories (e.g., recovering parametric types in a dependent setting).

2 Graded Modal Dependent Type Theory

The syntax of GRTT is that of a standard Martin-Löf type theory, extended with a graded modality and grades annotating binders of dependent function types: $(x :_{(s,r)} A) \to B$. Here, s and r range over the elements of a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$, where $+$ and $*$ are monotonic with respect to $\sqsubseteq$. Typing judgments in GRTT have the form:

$$(\Delta \mid \sigma_1 \mid \sigma_2) \odot \Gamma \vdash t : A$$

where the usual typing context Γ is treated as a vector, and Δ, σ_1, and σ_2 are vectors of the same size as Γ. Given $\Gamma[i]$ is an assumption $x : B$, then $\sigma_1[i] \in \mathcal{R}$ and $\sigma_2[i] \in \mathcal{R}$ are grades explaining x's usage in t (the subject) and A (the subject's type) respectively. Then $\Delta[i]$ is a vector of grades, of size i, which explains how each assumption prior to x is used in the formation of x's type, B. We refer to Δ as a *context grade vector*, and σ_1 and σ_2 as *grade vectors*.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Consider the semiring $(\mathbb{N}, \times, 1, +, 0, \equiv)$, capturing exact usage of variables, then the body of the polymorphic identity is typed: $((), (1) \mid 0, 1 \mid 1, 0) \odot A : \mathsf{Type}, x : A \vdash x : A$. Here, $\Delta = ((), (1))$ (a vector of vectors) explains that there are no assumptions prior to A, and A is used once (grade 1) in the formation of x's type in the context. Then $\sigma_1 = (0, 1)$ and $\sigma_2 = (1, 0)$ explain that A and x are used 0 and 1 times in the subject, and 1 and 0 times in the subject's type, respectively. Figure 1 shows selected typing rules of GRTT.

VAR introduces variables. We see two copies of σ (the dependencies used to form A) in the conclusion of the rule: one copy is used to type x in the context, by extending Δ; and the other is used to account for the type-level usage of A. The notation $\mathbf{0}$ (promotion of 0 to a vector of appropriate size) indicates that everything prior to x in the context should be associated with a 0 subject grade, as they are unused in the subject. The 1 and 0 grades denote the presence of x in the subject, and the absence of x in the subject type.

WEAK *weakens* a context with an *irrelevant* assumption x, by typing x in the context, and marking x with 0 grades. TYPE types universes under the empty context ($\emptyset$), using an inductive hierarchy [23] with ordering $<$. We capture the notion of *approximation* (e.g., an assumption that is used at most zero times is also used at most once) in the $\sqsubseteq$ rule. The $\sqsubseteq$ relation is lifted to grade and context grade vectors.

$$\frac{(\Delta \mid \sigma \mid \mathbf{0}) \odot \Gamma \vdash A : \mathsf{Type}_l}{(\Delta, \sigma \mid \mathbf{0}, 1 \mid \sigma, 0) \odot \Gamma, x : A \vdash x : A}\ \textsc{Var}$$

$$\frac{(\Delta \mid \sigma_1 \mid \sigma_2) \odot \Gamma \vdash t : A \quad (\Delta \mid \sigma_3 \mid \mathbf{0}) \odot \Gamma \vdash B : \mathsf{Type}_l}{(\Delta, \sigma_3 \mid \sigma_1, 0 \mid \sigma_2, 0) \odot \Gamma, x : B \vdash t : A}\ \textsc{Weak}$$

$$\frac{\begin{array}{c}(\Delta \mid \sigma_1 \mid \sigma_2) \odot \Gamma \vdash t : A \\ \Delta \sqsubseteq \Delta' \quad \sigma_1 \sqsubseteq \sigma_1' \quad \sigma_2 \sqsubseteq \sigma_2'\end{array}}{(\Delta' \mid \sigma_1' \mid \sigma_2') \odot \Gamma \vdash t : A}\ \sqsubseteq \qquad \frac{l_1 < l_2}{\emptyset \vdash \mathsf{Type}_{l_1} : \mathsf{Type}_{l_2}}\ \textsc{Type}$$

$$\frac{\begin{array}{c}(\Delta \mid \sigma_1 \mid \mathbf{0}) \odot \Gamma \vdash A : \mathsf{Type}_{l_1} \\ (\Delta, \sigma_1 \mid \sigma_2, r \mid \mathbf{0}) \odot \Gamma, x : A \vdash B : \mathsf{Type}_{l_2}\end{array}}{(\Delta \mid \sigma_1 + \sigma_2 \mid \mathbf{0}) \odot \Gamma \vdash (x :_{(s,r)} A) \to B : \mathsf{Type}_{l_1 \sqcup l_2}}\ \to$$

$$\frac{(\Delta, \sigma_1 \mid \sigma_2, s \mid \sigma_3, r) \odot \Gamma, x : A \vdash t : B}{(\Delta \mid \sigma_2 \mid \sigma_1 + \sigma_3) \odot \Gamma \vdash \lambda x.t : (x :_{(s,r)} A) \to B}\ \lambda_i$$

$$\frac{\begin{array}{c}(\Delta \mid \sigma_2 \mid \sigma_1 + \sigma_3) \odot \Gamma \vdash t_1 : (x :_{(s,r)} A) \to B \\ (\Delta \mid \sigma_4 \mid \sigma_1) \odot \Gamma \vdash t_2 : A\end{array}}{(\Delta \mid \sigma_2 + s * \sigma_4 \mid \sigma_3 + r * \sigma_4) \odot \Gamma \vdash t_1\ t_2 : [t_2/x]B}\ \lambda_e$$

$$\frac{(\Delta \mid \sigma \mid \mathbf{0}) \odot \Gamma \vdash A : \mathsf{Type}_l}{(\Delta \mid \sigma \mid \mathbf{0}) \odot \Gamma \vdash \Box_{(s,r)} A : \mathsf{Type}_l}\ \Box$$

$$\frac{(\Delta \mid \sigma_1 \mid \sigma_2) \odot \Gamma \vdash t : A}{(\Delta \mid s * \sigma_1 \mid \sigma_2 + r * \sigma_1) \odot \Gamma \vdash \Box t : \Box_{(s,r)} A}\ \Box_i$$

$$\frac{\begin{array}{c}(\Delta \mid \sigma_1 \mid \sigma_2 + \sigma_5) \odot \Gamma \vdash t_1 : \Box_{(s,r)} A \\ (\Delta, \sigma_5 \mid \sigma_3, s \mid \sigma_4, r) \odot \Gamma, x : A \vdash t_2 : B\end{array}}{(\Delta \mid \sigma_1 + \sigma_3 \mid \sigma_2 + \sigma_4) \odot \Gamma \vdash \mathsf{let}\ \Box x = t_1\ \mathsf{in}\ t_2 : \mathsf{let}\ \Box x = t_1\ \mathsf{in}\ B}\ \Box_e$$

Figure 1. Typing for GRTT

The $\to$ rule shows that in introducing a dependent function type, the dependencies of A and B are *contracted* by the operation $\sigma_1 + \sigma_2$ (vector addition using the + of the semiring). The usage of x in B is internalised as r in the binder. The grade s is arbitrary. λ_i introduces functions. The usage of x in t and B is described by grades s and r which are then captured in the binder. λ_e shows that to eliminate a function through application, the resources used to form the argument must be scaled by the amount specified in the binder.

Graded binders alone do not allow us to consider that different subparts of a term might be used in different ways, e.g., computing the length of a list ignores the elements, and projecting from a pair discards one component. We therefore introduce a *graded modality*, which allows us to capture the notion of local inspection on data, and allows usage information to be internalised to types. Our modality is in the style of Orchard et al. [22], but is *double-indexed*, allowing us to capture usage information at both the computational *and* type levels. The type former rule ($\Box$) for graded modal types is straightforward. The $\Box_i$ rule shows that we form a value $\Box t$ of type $\Box_{(s,r)} A$ by *scaling* the grades required to form t (of type A) by s and r, and providing these at the subject and subject-type levels, respectively. Finally, the $\Box_e$ rule shows that to eliminate a value of type $\Box_{(s,r)} A$, we need to say how to form a value under an assumption of type A that can be used with s-usage in the subject, and r-usage in the subject's type. Combining graded binders and graded modalities makes for a highly expressive system, allowing precise usage information on compound data.

Figure 2 shows how we define a projection function.

3 Discussion

There has been a recent resurgence in linear types, e.g., with linearity influencing the borrowing system of Rust [3], work to integrate linearity into Haskell via a grading-style approach [4], and session types bringing linearity into the focus of everyday programming [13]. Thus, now is a good time to bring dependent types more clearly into the linear types story. This work constitutes a step towards a dependently-typed language with comprehensive resource tracking. Rather than biasing towards the computational level, we advance the start-of-the-art via quantitative tracking at all layers.

This is work in progress and we are developing an implementation (called Gerty). Further work is to add equality types (in the style of [21]), and coproducts and the resulting control-flow analysis. Our aim is to enable a new generation of programming languages and proof assistants that put expressive resource reasoning at programmers' fingertips.

$$\begin{array}{l}\mathsf{proj1} : (a :_{(0,2)} \mathsf{Type})\ (b :_{(0,1)} \mathsf{Type})\ (x :_{(1,1)} (\Box_{(1,0)} a,\ \Box_{(0,0)} b)) \\ \qquad \to \mathsf{let}\ (\Box l, \Box r) = x\ \mathsf{in}\ a \\ \mathsf{proj1}\ a\ b\ x = \mathsf{let}\ (\Box l, \Box r) = x\ \mathsf{in}\ l\end{array}$$

Figure 2. GRTT projection function

References

[1] Andreas Abel. 2018. Resourceful Dependent Types. In *24th International Conference on Types for Proofs and Programs, Abstracts*.

[2] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 56–65. https://doi.org/10.1145/3209108.3209189

[3] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System programming in Rust: Beyond safety. *ACM SIGOPS Operating Systems Review* 51, 1 (2017), 94–99.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 5.

[5] Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive families need not store their indices. In *International Workshop on Types for Proofs and Programs*. Springer, 115–129.

[6] Flavien Breuvart and Michele Pagani. 2015. Modelling Coeffects in the Relational Semantics of Linear Logic. In *CSL*.

[7] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *ESOP*. 351–370.

[8] Iliano Cervesato and Frank Pfenning. 2002. A linear logical framework. *Information and Computation* 179, 1 (2002), 19–75.

[9] U. Dal Lago and M. Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In IEEE LICS '11. 133–142.

[10] Arthur Azevedo De Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, 5.

[11] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *POPL*. 357–370. http://dl.acm.org/citation.cfm?id=2429113

[12] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

[13] Simon J. Gay. 1995. *Linear Types for Communicating Processes*. Ph.D. Dissertation. University of London.

[14] Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *ESOP*. 331–350.

[15] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

[16] 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.

[17] Martin Hofmann. 2000. A type system for bounded space and functional in-place update. In *European Symposium on Programming*. Springer, 165–179.

[18] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating linear and dependent types. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 17–30.

[19] Zhaohui Luo and Yu Zhang. 2016. A Linear Dependent Type Theory. *Types for Proofs and Programs (TYPES 2016), Novi Sad* (2016).

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

[21] Ulf Norell. 2007. Towards a practical programming language based on dependent type theory.

[22] 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

[23] Erik Palmgren. 1998. On universes in type theory. *Twenty-five years of constructive type theory* 36 (1998), 191–204.

[24] 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 (ICFP '14)*. ACM, 123–135. https://doi.org/10.1145/2628136.2628160

[25] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North.