Deriving distributive laws for graded linear types

Jack Hughes
School of Computing, University of Kent

Dominic Orchard
School of Computing, University of Kent

The recent idea of graded modal types provides a framework for extending type theories with quantitative reasoning principles. The Granule language explores this idea in the context of linear types. In this practical setting, when composing programs it is often necessary to distribute algebraic data types over graded modalities, and vice versa. We describe how to automatically derive these distributive laws, and explore pattern matching semantics in the context of graded, linear types. This forms part of a larger work on program synthesis and practical programming with graded modal linear types.

Context Bounded Linear Logic (BLL) provides a family of modalities $!_r A$ where $r \in \mathbb{N}$ captures the maximum number of times r that a value A can be used [5]. The proof rules of BLL use these indices to track upper-bounds on non-linear use. Recently, various works have generalised BLL to an arbitrary semiring [1, 3, 8], providing a unified approach to capturing program properties. This has been further generalised by the recent notion of *graded modal* types, which encompasses BLL, its generalisations, and other graded structures like graded monads [7]. The programming language Granule provides a language for exploring graded model types, combining them with indexed and linear types [4].

When programming with graded modal types, there is often a need to distribute a graded modality over a type, and vice versa, in order to compose pre-existing code. For example, imagine a programmer has a non-empty list and wants to create a pair from the head element. The `head` operation is typed: `head : ∀ {a : Type, n : Nat} . (Vec (n + 1) a) [0..1] ⊸ a` which has a graded modal input (the modal operator is postfix) with grade `0..1` meaning the input vector is used 0 or 1 times: the head element is used once (linearly) for the return but the tail is discarded. This head element can then be copied if it has this capability via a graded modality. A value of type `(Vec (n + 1) (a [2])) [0..1]` permits:

```
copyHead' : ∀ {a : Type, n : Nat} . (Vec (n + 1) (a [2])) [0..1] ⊸ (a, a)
copyHead' xs = let [y] = head xs in (y, y) -- [y] unboxes (a [2]) to y:a which can be used twice
```

However, what if we are in a programming context where we have a value `Vec (n + 1) a` with no graded modality on the type `a`? We can employ two idioms here: (i) take a value of type `(Vec (n + 1) a) [0..2]` and split its modality in two: `(Vec (n + 1) a) [2] [0..1]` (ii) then use a *distributive law* of the inner graded modality `[2]` to "push" it inside the vector to get `(Vec (n + 1) (a [2])) [0..1]`. Both this distributive law and its dual are provided by Granule's standard library, typed:

```
pushVec : ∀ {a : Type, n : Nat, k : Coeffect, c : k} . {c ⩾ 1} ⇒ (Vec n a) [c] ⊸ Vec n (a [c])
pullVec : ∀ {a : Type, n : Nat, k : Coeffect, c : k} . Vec n (a [c]) ⊸ (Vec n a) [c]
```

where `pushVec` describes a distributive law of the vector type over any graded necessity modality (whose grade types are called "coeffects"). The part of the type which reads `c ⩾ 1` explains that, since we are producing a linear value, there is some consumption going on which must be witnessed in the graded `c`.

Using `pushVec` we can thus write the following to duplicate the head element of a vector:

```
copyHead : ∀ {a : Type, n : Nat} . (Vec (n + 1) a) [0..2] ⊸ (a, a)
copyHead [xb] = let xbb = [[xb]] in let [y] = head (let [xb] = xbb in [pushVec xb]) in (y, y)
```

Similar "push" and "pull" operations are defined throughout the standard library for sums, products, and other ADTs such as lists. There is a general pattern to these distributive operations which allows their programs to be automatically synthesised from their type structure alone by a simple inductive procedure. We describe the procedure, applying a generic programming methodology [6].

To appear in EPTCS.

Core typing We show a simplified monomorphic subset of Granule and its *semiring graded necessities* [7] where for a pre-ordered semiring $(\mathscr{R}, *, 1, +, 0, \sqsubseteq)$ there is a family of types $\{\Box_r A\}_{r \in \mathscr{R}}$. Typing contexts allow linear or graded assumptions $\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, y : [A]_c$ where y is an assumption graded by c drawn from some semiring. Syntax and typing is that of the linear λ-calculus, plus:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{DER} \qquad \frac{\Gamma \vdash t : A}{\Gamma + [\Delta]_0 \vdash t : A}\ \text{WEAK} \qquad \frac{[\Gamma] \vdash t : A}{r * [\Gamma] \vdash [t] : \Box_r A}\ \text{PR}$$

$$\frac{\Gamma \vdash t : A \quad \cdot \vdash p_i : A \rhd \Delta_i \quad \Gamma', \Delta_i \vdash t_i : B}{\Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ..; p_n \mapsto t_n : B}\ \text{CASE}$$

where $\Gamma + \Gamma'$ is defined via semiring addition $(\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) = (\Gamma + \Gamma'), x : [A]_{r+s}$ for contexts whose linear assumptions are disjoint. Graded modalities are introduced by PR, scaling graded assumptions in Γ via semiring multiplication ($[\Gamma]$ denotes contexts with only graded assumptions). Graded modal elimination is by **case** with an *unboxing* pattern ([PBOX]). Patterns p are typed by $r : ?R \vdash p : A \rhd \Gamma$ where Γ is the context of bindings in p, and $r : ?R \ ::= \cdot \mid r : R$ denotes if a pattern is inside an unboxing pattern:

$$\frac{}{\cdot \vdash x : A \rhd x : A}\ \text{PVAR} \qquad \frac{\cdot \vdash p_i : B_i \rhd \Gamma_i}{\cdot \vdash C\, p_1 .. p_n : A \rhd \Gamma_1, .., \Gamma_n}\ \text{PCON} \qquad \frac{r : R \vdash p : A \rhd \Gamma}{\cdot \vdash [p] : \Box_r A \rhd \Gamma}\ \text{PBOX}$$

$$\frac{}{r : R \vdash x : A \rhd x : [A]_{r:R}}\ [\text{PVAR}] \qquad \frac{r : R \vdash p_i : B_i \rhd \Gamma_i \quad |A| \geq 1 \Rightarrow 1 \sqsubseteq r}{r : R \vdash C\, p_1 .. p_n : A \rhd \Gamma_1, .., \Gamma_n}\ [\text{PCON}] \qquad \frac{0 \sqsubseteq r}{r : R \vdash \_ : A \rhd \emptyset}\ [\text{PWILD}]$$

where $(C : B_1 \multimap \ldots \multimap B_n \multimap A)$ in PCON and [PCON], and multiplicative products and additive sums are then by $(,) : A \multimap B \multimap A \otimes B$, $\mathsf{inl} : A \multimap A \oplus B$, and $\mathsf{inr} : B \multimap A \oplus B \in C$ for all A, B.

The natural numbers semiring with $\sqsubseteq = (\equiv)$ provides *exact usage* and with $\sqsubseteq = \leq$ provides BLL-style grading by an upper-bound on usage. Granule also provides graded modalities with semirings of security-level lattices, intervals of lower-and-upper bound use, and "sensitivities" akin to *DFuzz* [2].

Semantics of consumption in pattern typing Wildcard patterns are only allowed inside a graded modality ([PWILD]), requiring $0 \sqsubseteq r$, i.e., r can approximate 0 (where 0 denotes weakening). For a type A with more than one constructor ($|A| \geq 1$), pattern matching its constructors underneath an r-graded box requires $1 \sqsubseteq r$. For example, eliminating sums inside an r-graded box $\Box_r(A \oplus B)$ requires $1 \sqsubseteq r$ as distinguishing inl or inr constitutes a *consumption* which reveals information. By contrast, a type with only one constructor cannot convey any information by its constructor and so matching on it is not counted as a consumption: eliminating $\Box_r(A \otimes B)$ places no requirements on r. This has implications for *push* and *pull* and is a change to Granule here (where [7] lacked the precondition $|A| \geq 1$ in [PCON]).

Automatically deriving *push* and *pull* Let $\mathsf{F} : \mathsf{Type}^n \to \mathsf{Type}$ be an n-ary type constructor (which may be a recursive type via $\mu X.A$). We write $\mathsf{F}\overline{A_i}$ for the application of F to argument types A_i for all $1 \leq i \leq n$.

We automatically calculate *push* and *pull* for F applied to n type variables $\overline{\alpha_i}$ as the operations:

$$[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \mathsf{F}(\overline{\Box_r \alpha_i}) \qquad\qquad [\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}} : \mathsf{F}\ (\overline{\Box_{r_i} \alpha_i}) \multimap \Box_{\bigwedge_{i=1}^n r_i}(\mathsf{F}\ \overline{\alpha_i})$$

where for $[\![-]\!]_{\mathsf{push}}$ we require $1 \sqsubseteq r$ *iff* $|\mathsf{F}\overline{\alpha_i}| \geq 1$ due to the [PCON] rule (e.g., if F contains a sum). For closed types A, let $[\![A]\!]_{\mathsf{push}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{push}} z$ and $[\![A]\!]_{\mathsf{pull}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{pull}} z$ where (for recursion variables Σ):

$$\begin{aligned}
[\![1]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ [()] \to () \\
[\![X]\!]^{\Sigma}_{\mathsf{push}}\ z &= \Sigma(X)\ z \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\ z &= z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x];\ [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]) \\
[\![A \multimap B]\!]_{\mathsf{push}}\ z &= \lambda y.\textbf{case}\ z\ \textbf{of}\ [f] \to \textbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \textbf{of}\ [u] \to [\![B]\!]^{\Sigma}_{\mathsf{push}}[(f\ u)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}_{\mathsf{push}}\ \textbf{in}\ f\ z
\end{aligned}$$

$$\begin{aligned}
[\![1]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ () \to [()] \\
[\![X]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \Sigma(X)\ z \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{pull}}\ z &= z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ \mathsf{inl}\ x \to \textbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x\ \textbf{of}\ [u] \to [\mathsf{inl}\ u]; \\
&\qquad\qquad \mathsf{inr}\ y \to \textbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \textbf{of}\ [v] \to [\mathsf{inr}\ v] \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ (x,y) \to \textbf{case}\ ([\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y)\ \textbf{of}\ ([u],[v]) \to [(u,v)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.A[\overrightarrow{\Box_{r_i} \alpha_i / \alpha_i}] \multimap \Box_{\bigwedge_{i=1}^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \textbf{in}\ f\ z
\end{aligned}$$

Neither operation can be applied to graded modalities themselves. Unlike *pull*, the *push* operation can be derived for function types, with a contravariant use of *pull*.

Proposition 1 (Pull is right inverse to push)**.** $[\![[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}]\!]_{\mathsf{push}} = id : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \mathsf{F}(\overline{\Box_r \alpha_i})$

Proposition 2 (Pull is left inverse to push, under type restriction)**.** $[\![[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}]\!]_{\mathsf{pull}} = id : \mathsf{F}(\overline{\Box_r \alpha_i}) \multimap \Box_r \mathsf{F}\overline{\alpha_i}$

Next steps (*i*) Prove that these operations are distributive laws with respect to the graded comonad structure of $\Box_r A$; (*ii*) Complete the implementation in Granule (adapting the approach to user-defined (G)ADTs); (*iii*) Derive other useful generic combinators such as the *copyShape* : $\mathsf{F}\ \overline{\alpha_i} \multimap (\mathsf{F}\ 1 \otimes \mathsf{F}\ \overline{\alpha_i})$.

References

[1] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *European Symposium on Programming Languages and Systems*, pages 351–370. Springer, 2014.

[2] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 357–370, 2013.

[3] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Programming Languages and Systems*, pages 331–350. Springer, 2014.

[4] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

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

[6] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 2000.

[7] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *PACMPL*, 3(ICFP):110:1–110:30, 2019.

[8] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 123–135. ACM, 2014.