Deriving Distributive Laws for Graded Linear Types

Jack Hughes
School of Computing, University of Kent

Michael Vollmer
School of Computing, University of Kent

Dominic Orchard
School of Computing, University of Kent

The recent notion of graded modal types provides a framework for extending type theories with fine-grained data-flow reasoning. The Granule language explores this idea in the context of linear types. In this practical setting, we observe that the presence of graded modal types can introduce an additional impediment when programming: when composing programs, it is often necessary to 'distribute' data types over graded modalities, and vice versa. In this paper, we show how to automatically derive these distributive laws as combinators for programming. We discuss the implementation and use of this automated deriving procedure in Granule, providing easy access to these distributive combinators. This work is also applicable to Linear Haskell (which retrofits Haskell with linear types via grading) and we apply our technique there to provide the same automatically derived combinators. Along the way, we discuss interesting considerations for pattern matching analysis via graded linear types. Lastly, we show how other useful structural combinators can also be automatically derived.

1 Introduction

Linear type systems capture and enforce the idea that some data travels a *linear* dataflow path through a program, being consumed exactly once. This is enforced by disallowing the structural rules of weakening and contraction on variables carrying such values. Non-linear dataflow is then typically captured by a modality $!A$ characterising values/variables on which all structural rules are permitted [12]. This binary characterisation (linear *versus* non-linear) is made more fine-grained in Bounded Linear Logic (BLL) via a family of modalities $!_r A$ where r is a polynomial term in $\mathbb{N}$ capturing the maximum number of times r that a value A can be used [13]. The proof rules of BLL track upper-bounds on non-linear use via these indices. Various works have generalised the indices of BLL's modalities to arbitrary semirings [6, 11, 23] (often referred to as *coeffect* systems). These generalised systems provide a unified way to capture various program properties relating to dataflow via *graded necessity modalities* $\Box_r A$ where r is drawn from a semiring. The functional programming language Granule[1] provides a vehicle for exploring graded modal types (both graded necessity and graded possibility, which can model side effects [19]) alongside linear and indexed types (GADTs) [22]. There are various similar systems in the recent literature: Abel and Bernardy give an alternate graded modal calculus [1] and Linear Haskell retrofits linearity onto the Haskell type system [3] via a graded type system akin to the coeffect calculus of Petricek et al. [23]. The popularity of graded types is growing and their practical implications are being explored in research languages (e.g. Granule), and popular functional languages (e.g. Haskell [3] and Idris 2 [4]).

When programming with graded modal types, we have observed there is often a need to 'distribute' a graded modality over a type, and vice versa, in order to compose programs. That is, we may find ourselves in possession of a $\Box_r(\mathsf{F}\alpha)$ value (for some parametric data type F) which needs to be passed to a pre-existing function (of our own codebase or a library) which requires a $\mathsf{F}(\Box_r\alpha)$ value, or perhaps vice

[1] `https://granule-project.github.io/`

U. Dal Lago and V. de Paiva (Eds.) Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications.
EPTCS 353, 2021, pp. 109–131, doi:10.4204/EPTCS.353.6

versa. A *distributive law* (in the categorical sense, e.g., [27]) provides a conversion from one to the other. In this paper, we present a procedure to automatically synthesise these distributive operators, applying a generic programming methodology [14] to compute these operations given the base type (e.g., $\mathsf{F}\alpha$ in the above description). This serves to ease the use of graded modal types in practice, removing boilerplate code by automatically generating these 'interfacing functions' on-demand, for user-defined data types as well as built-in types.

Throughout, we refer to distributive laws of the form $\Box_r(\mathsf{F}\alpha) \to \mathsf{F}(\Box_r\alpha)$ as *push* operations (as they 'push' the graded modality inside the type constructor F), and dually $\mathsf{F}(\Box_r\alpha) \to \Box_r(\mathsf{F}\alpha)$ as *pull* operations (as they 'pull' the graded modality outside the type constructor F).

The primary contributions of this paper are then:

- an overview of the application of distributive laws in the context of graded modal types;
- an automatic procedure for calculating distributive laws from types and a formal analysis of their properties;
- a realisation of this approach in both Granule (embedded into the compiler) and Linear Haskell (expressed within the language itself, leveraging Haskell's advanced features);
- and derivations of related combinators for structural use of values in a graded linear setting.

Along the way, we also analyse choices made around the typed-analysis of pattern matching in various recent works on graded modal types. Through the comparison between Granule and Linear Haskell, we also highlight ways in which Linear Haskell could be made more general and flexible in the future.

Section 2 defines a core calculus GrMini^P with linear and graded modal types which provides an idealised, simply-typed subset of Granule with which we develop the core contribution. Section 3 gives the procedures for deriving *push* and *pull* operators for the core calculus, and verifies that these are distributive laws of endofunctors over the $\Box_r$ graded comonadic modality. Section 4 describes the details of how these procedures are realised in the Granule language. Section 5 relates this work to Linear Haskell, and demonstrates how the *push* and *pull* combinators for user-defined data types may be automatically generated at compile-time using Template Haskell. Section 6 gives a comparison of the recent literature with regards the typed (graded) analysis of pattern matching, which is germane to the typing and derivation of our distributive laws. Section 7 covers how other structural combinators for Granule and Linear Haskell may be derived. Finally, Section 8 discusses more related and future work.

We start with an extended motivating example typifying the kind of software engineering impedance problem that distributive laws solve. We use Granule since it is the main vehicle for the developments here, and introduce some of the key concepts of graded modal types (in a linear context) along the way.

1.1 Motivating Example

Consider the situation of projecting the first element of a pair. In Granule, this first-projection is defined and typed as the following polymorphic function (whose syntax is reminiscent of Haskell or ML):

```
fst : ∀ {a b : Type} . (a, b [0]) → a
fst (x, [y]) = x
```

Linearity is the default, so this represents a linear function applied to linear values.[2] However, the second component of the pair has a *graded modal type*, written `b [0]`, which means that we can use the value "inside" the graded modality 0 times by first 'unboxing' this capability via the pattern match `[y]` which

[2]Granule uses $\to$ syntax rather than $\multimap$ for linear functions for the sake of familiarity with standard functional languages

allows weakening to be applied in the body to discard `y` of type `b`. In calculus of Section 2, we denote '`b [0]`' as the type $\Box_0 b$ (Granule's graded modalities are written postfix with the 'grade' inside the box).

The type for `fst` is however somewhat restrictive: what if we are trying to use such a function with a value (call it `myPair`) whose type is not of the form `(a, b [0])` but rather `(a, b) [r]` for some grading term `r` which permits weakening? Such a situation readily arises when we are composing functional code, say between libraries or between a library and user code. In this situation, `fst myPair` is ill-typed. Instead, we could define a different first projection function for use with `myPair : (a, b) [r]` as:

```
fst' : ∀ {a b : Type, s : Semiring, r : s} . {0 ⩽ r} ⇒ (a, b) [r] → a
fst' [(x, y)] = x
```

This implementation uses various language features of Granule to make it as general as possible. Firstly, the function is polymorphic in the grade `r` and in the semiring `s` of which `r` is an element. Next, a refinement constraint `0 ⩽ r` specifies that by the preordering $\leqslant$ associated with the semiring `s`, that `0` is approximated by `r` (essentially, that `r` permits weakening). The rest of the type and implementation looks more familiar for computing a first projection, but now the graded modality is over the entire pair.

From a software engineering perspective, it is cumbersome to create alternate versions of generic combinators every time we are in a slightly different situation with regards the position of a graded modality. Fortunately, this is an example to which a general *distributive law* can be deployed. In this case, we could define the following distributive law of graded modalities over products, call it `pushPair`:

```
pushPair : ∀ {a b : Type, s : Semiring, r : s} . (a, b) [r] → (a [r], b [r])
pushPair [(x, y)] = ([x], [y])
```

This 'pushes' the graded modality `r` into the pair (via pattern matching on the modality and the pair inside it, and then reintroducing the modality on the right hand side via `[x]` and `[y]`), distributing the graded modality to each component. Given this combinator, we can now apply `fst (pushPair myPair)` to yield a value of type `a [r]`, on which we can then apply the Granule standard library function `extract`, defined:

```
extract : ∀ {a : Type, s : Semiring, r : s} . {(1 : s) ⩽ r} ⇒ a [r] → a
extract [x] = x
```

to get the original `a` value we desired:

```
extract (fst (pushPair myPair)) : a
```

The `pushPair` function could be provided by the standard library, and thus we have not had to write any specialised combinators ourselves: we have applied supplied combinators to solve the problem.

Now imagine we have introduced some custom data type `List` on which we have a *map* function:

```
data List a = Cons a (List a) | Nil

map : ∀ {a b : Type} . (a → b) [0..∞] → List a → List b
map [f] Nil = Nil;
map [f] (Cons x xs) = Cons (f x) (map [f] xs)
```

Note that, via a graded modality, the type of `map` specifies that the parameter function, of type `a → b` is non-linear, used between 0 and ∞ times. Imagine now we have a value `myPairList : (List (a, b)) [r]` and we want to map first projection over it. But `fst` expects `(a, b [0])` and even with `pushPair` we require `(a, b) [r]`. *We need another distributive law*, this time of the graded modality over the `List` data type. Since `List` was user-defined, we now have to roll our own `pushList` operation, and so we are back to having to make specialised combinators for our data types.

The crux of this paper is that such distributive laws can be automatically calculated given the definition of a type. With our Granule implementation of this approach (Section 4), we can then solve this

combination problem via the following composition of combinators:

```
map (extract . fst . push @(,)) (push @List myPairList) : List a
```

where the `push` operations are written with their base type via `@` (a type application) and whose definitions and types are automatically generated during type checking. Thus the `push` operation is a *data-type generic function* [14]. This generic function is defined inductively over the structure of types, thus a programmer can introduce a new user-defined algebraic data type and have the implementation of the generic distributive law derived automatically. This reduces both the initial and future effort (e.g., if an ADT definition changes or new ADTs are introduced).

Dual to the above, there are situations where a programmer may wish to *pull* a graded modality out of a structure. This is possible with a dual distributive law, which could be written by hand as:

```
pullPair : ∀ {a b : Type, s : Semiring, m n : s} . (a [n], b [m]) → (a, b) [n ⊓ m]
pullPair ([x], [y]) = [(x, y)]
```

Note that the resulting grade is defined by the greatest-lower bound (meet) of `n` and `m`, if it exists as defined by a preorder for semiring `s` (that is, $\sqcap$ is not a total operation). This allows some flexibility in the use of the *pull* operation when grades differ in different components but have a greatest-lower bound which can be 'pulled out'. Our approach also allows such operations to be generically derived.

2 Core Calculus, GrMini^P

We define here a simplified monomorphic subset of Granule, which we call the GrMini^P calculus.[3] This calculus extends the linear λ-calculus with a *semiring graded necessity modality* [22] where for a preordered semiring $(\mathscr{R}, *, 1, +, 0, \sqsubseteq)$ (that is, a semiring with a pre-order $\sqsubseteq$ where $+$ and $*$ are monotonic with respect to $\sqsubseteq$) there is a family of types $\{\Box_r A\}_{r \in \mathscr{R}}$. Granule allows multiple graded modalities (and thus multiple grading semirings) to be used simultaneously within a program, but we focus here on just one graded modality, parameterising the language. We also include notions of data constructor and their elimination via **case** expressions as a way to unify the handling of regular type constructors.

The syntax of GrMini^P terms and types is given by:

$$
\begin{array}{rll}
t ::= & x \mid t_1\, t_2 \mid \lambda x.t \mid [t] \mid C\, t_0 \ldots t_n \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n \mid \textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2 & \text{(terms)} \\
p ::= & x \mid \_ \mid [p] \mid C\, p_0 \ldots p_n & \text{(patterns)} \\
A, B ::= & A \multimap B \mid \alpha \mid A \otimes B \mid A \oplus B \mid \mathbf{1} \mid \Box_r A \mid \mu X.A \mid X & \text{(types)} \\
C ::= & \mathsf{unit} \mid \mathsf{inl} \mid \mathsf{inr} \mid (\,,\,) & \text{(data constructors)}
\end{array}
$$

Terms of GrMini^P consist of those of the linear λ-calculus, plus a *promotion* construct $[t]$ for introducing a graded modality, data constructor introduction $C\, t_0 \ldots t_n$ and elimination (**case**), as well as recursive let bindings (**letrec**). We will give more insights into the syntax via their typing rules. Types comprise linear function types, type variables (α), multiplicative products, additive sums, unit types, graded modal types, recursive types, and recursion variables (X) (which must be guarded by a μ binding). Type variables are used in our derivation procedure but are treated here as constants by the core calculus, without any rules relating to their binding or unification.

[3] GrMini^P is akin to the GrMini calculus given by Orchard et al. [22] but we include here the notions of data type constructors, eliminations, and pattern matching since these are important to the development here.

$$\frac{}{x:A\vdash x:A}\ \textsc{var} \qquad \frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash \lambda x.t:A\multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1\vdash t_1:A\multimap B \qquad \Gamma_2\vdash t_2:A}{\Gamma_1+\Gamma_2\vdash t_1\,t_2:B}\ \textsc{app}$$

$$\frac{\Gamma,x:A\vdash t:B}{\Gamma,x:[A]_1\vdash t:B}\ \textsc{der} \qquad \frac{\Gamma\vdash t:A}{\Gamma,[\Delta]_0\vdash t:A}\ \textsc{weak} \qquad \frac{[\Gamma]\vdash t:A}{r*[\Gamma]\vdash [t]:\Box_r A}\ \textsc{pr}$$

$$\frac{\Gamma,x:[A]_r,\Gamma'\vdash t:A \qquad r\sqsubseteq s}{\Gamma,x:[A]_s,\Gamma'\vdash t:A}\ \textsc{approx} \qquad \frac{\Gamma,x:A\vdash t_1:A \qquad \Gamma',x:A\vdash t_2:B}{\Gamma+\Gamma'\vdash \textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2:B}\ \textsc{letrec}$$

$$\frac{(C:B_1\multimap\ldots\multimap B_n\multimap A)\in \mathrm{D}}{\emptyset\ \vdash\ C:B_1\multimap\ldots\multimap B_n\multimap A}\ \textsc{con} \qquad \frac{\Gamma\vdash t:A \quad \cdot\vdash p_i:A\rhd\Delta_i \quad \Gamma',\Delta_i\vdash t_i:B}{\Gamma+\Gamma'\vdash \textbf{case}\ t\ \textbf{of}\ p_1\mapsto t_1;..;p_n\mapsto t_n:B}\ \textsc{case}$$

Figure 1: Typing rules for GrMini^P

2.1 Typing

Typing is via judgments of the form $\Gamma\vdash t:A$, assigning a type A to term t under the context Γ. Typing contexts Γ contain linear or graded assumptions given by the grammar:

$$\Gamma ::= \emptyset \mid \Gamma,x:A \mid \Gamma,y:[A]_r$$

where x is a linear assumption and where y is an assumption graded by r drawn from the preordered semiring (called a *graded assumption*). This delineation of linear and graded assumptions avoids issues with substitution under a promotion, following the technique of Terui's *discharged assumptions* [28]. Linear assumptions (including those of a graded modal type $\Box_r A$) must therefore be used exactly once, whilst graded assumptions may be used non-linearly subject to the constraint of r.

Figure 1 gives the typing rules. We briefly explain each rule in turn.

The VAR, ABS and APP rules follow the standard rules for the linear λ-calculus, modulo the *context addition* operation $\Gamma+\Gamma'$ which is used in the rules any time the contexts of two subterms need to be combined. Context addition acts as union on disjoint linear and graded assumptions but is defined via semiring addition when a graded assumption appears in both contexts:

$$(\Gamma,x:[A]_r)+(\Gamma',x:[A]_s)=(\Gamma+\Gamma'),x:[A]_{r+s}$$

Context addition $+$ is undefined in the case of contexts whose linear assumptions are not-disjoint, enforcing the lack of contraction on linear assumptions.

The next rules relate to grading. Structural weakening is provided by (WEAK) for assumptions graded by 0 and 'dereliction' (DER) converts a linear assumption to a graded assumption, graded by 1. Graded modalities are introduced by 'promotion' (PR), scaling graded assumptions in Γ via semiring multiplication defined on contexts containing only graded assumptions as:

$$r*\emptyset=\emptyset \qquad\qquad r*(\Gamma,x:[A]_s)=(r*\Gamma),x:[A]_{(r*s)}$$

In the (PR) rule, $[\Gamma]$ denotes contexts with only graded assumptions. The APPROX rule captures grade approximation, allowing a grade r in an assumption to be converted to another grade s, on the condition that r *is approximated by* s, via the relation $\sqsubseteq$ as provided by the semiring's preorder.

The LETREC rule provides recursive bindings in the standard way.

$$\frac{}{\cdot \vdash x : A \rhd x : A}\ \textsc{Pvar} \qquad \frac{\cdot \vdash p_i : B_i \rhd \Gamma_i}{\cdot \vdash C\,p_1\,..\,p_n : A \rhd \Gamma_1, .., \Gamma_n}\ \textsc{Pcon} \qquad \frac{r \vdash p : A \rhd \Gamma}{\cdot \vdash [p] : \Box_r A \rhd \Gamma}\ \textsc{Pbox}$$

$$\frac{}{r \vdash x : A \rhd x : [A]_r}\ [\textsc{Pvar}] \qquad \frac{r \vdash p_i : B_i \rhd \Gamma_i \qquad |A| > 1 \Rightarrow 1 \sqsubseteq r}{r \vdash C\,p_1\,..\,p_n : A \rhd \Gamma_1, .., \Gamma_n}\ [\textsc{Pcon}] \qquad \frac{0 \sqsubseteq r}{r \vdash \_ : A \rhd \emptyset}\ [\textsc{Pwild}]$$

Figure 2: Pattern typing rules for GrMini^P

Data constructors with zero or more arguments are introduced via the CON rule. Here, the constructors that concern us are units, products, and coproducts (sums), given by D, a global set of data constructors with their types, defined:

$$D = \{\mathsf{unit} : \mathbf{1}\} \cup \{(,) : A \multimap B \multimap A \otimes B \mid \forall A, B\} \cup \{\mathsf{inl} : A \multimap A \oplus B \mid \forall A, B\} \cup \{\mathsf{inr} : B \multimap A \oplus B \mid \forall A, B\}$$

Constructors are eliminated by pattern matching via the CASE rule. Patterns p are typed by judgments of the form $?r \vdash p : A \rhd \Delta$ meaning that a pattern p has type A and produces a context of typed binders Δ (used, e.g., in the typing of the case branches). The information to the left of the turnstile denotes optional grade information arising from being in an unboxing pattern and is syntactically defined as either:

$$?r \ ::= \cdot \mid r \qquad \text{(enclosing grade)}$$

where $\cdot$ means the present pattern is not nested inside an unboxing pattern and r that the present pattern is nested inside an unboxing pattern for a graded modality with grade r.

The rules of pattern typing are given in Figure 2. The rule (PBox) provides graded modal elimination (an 'unboxing' pattern), propagating grade information into the typing of the sub-pattern. Thus **case** t **of** $[p] \to t'$ can be used to eliminate a graded modal value. Variable patterns are typed via two rules depending on whether the variable occurs inside an unbox pattern ([Pvar]) or not (Pvar), with the [Pvar] rule producing a binding with the grade of the enclosing box's grade r. As with variable patterns, constructor patterns are split between rules for patterns which either occur inside an unboxing pattern or not. In the former case, the grade information is propagated to the subpattern(s), with the additional constraint that if there is more than one data constructor for the type A (written $|A| > 1$), then the grade r must approximate 1 (written $1 \sqsubseteq r$) as pattern matching incurs a usage to inspect the constructor (discussed further in Section 6). The operation $|A|$ counts the number of data constructors for a type:

$$|\mathbf{1}| = 1 \quad |A \multimap B| = 1 \quad |\Box_r A| = |A| \quad |A \oplus B| = 2(|A| + |B|) \quad |A \otimes B| = |A||B| \quad |\mu X.A| = |A[\mu X.A/X]|$$

and $|X|$ is undefined (or effectively 0) since we do not allow unguarded recursion variables in types. A type A must therefore involve a sum type for $|A| > 1$.

Since a wildcard pattern _ discards a value, this is only allowed inside an unboxing pattern where the enclosing grade permits weakening, captured via $0 \sqsubseteq r$ in rule [Pwild].

We now provide two examples of graded modalities. In addition to those shown here, Granule provides various other graded modalities, including grading by semirings of security-level lattices and "sensitivities" akin to *DFuzz* [9].

Example 2.1. The natural numbers semiring $(\mathbb{N}, *, 1, +, 0, \equiv)$ provides *exact usage* where the preorder is equality $\equiv$. Graded modalities in this semiring count the number of times a value is used. As a simple

example, we can define a function which copies its input value to produce a pair of values:

$$copy : \Box_2 A \multimap (A \otimes A)$$
$$copy = \lambda y.\textbf{case}\ y\ \textbf{of}\ [x] \to (x,x)$$

The capability to use a value of type A twice is captured by the graded modality. In the body of the abstraction, variable $y : \Box_2 A$ must be used linearly; y is used linearly in the **case** which eliminates the graded modality to yield a graded assumption $x : [A]_2$ which can be used twice to form the resulting pair.

Example 2.2. BLL-style grading with upper bounds on usages can be obtained by replacing the equality relation in the $\mathbb{N}$-semiring with a less-than-equal ordering, giving a notion of *approximation*. This is useful for programs that use variables differently in control-flow branches, e.g., when eliminating a sum type. This notion is further refined by the semiring of natural number intervals, which captures lower and upper bounds on usages. Intervals are given as pairs $\mathbb{N} \times \mathbb{N}$ written $r..s$, where r and s represent the lower and upper bounds respectively, with $r \leq s$, $0 = 0..0$ and $1 = 1..1$. Ordering is given by $(r..s) \leq (r'..s') = r' \leq r \wedge s \leq s'$. Using this, we can define the following function to eliminate a coproduct via a pair of functions each of which is used either 0 or 1 times:

$$\oplus_e : \Box_{[0..1]}(A \multimap C) \multimap \Box_{[0..1]}(B \multimap C) \multimap (A \oplus B) \multimap C$$
$$\oplus_e = \lambda f'.\lambda g'.\lambda z.\textbf{case}\ f'\ \textbf{of}\ [f] \to \textbf{case}\ g'\ \textbf{of}\ [g] \to \textbf{case}\ z\ \textbf{of}\ \mathsf{inl}\ x \mapsto f\,x; \mathsf{inr}\ y \to g\,y$$

2.2 Equational Theory

Figure 3 defines an equational theory for GrMini^P. The equational theory is typed, but for brevity we elide the typing for most cases since it follows exactly from the structure of the terms. The fully typed rules are provided in the appendix [17]. For those rules which are type restricted, we include the full typed-equality judgment here. The typability of these equations relies on previous work on the Granule language which proves that pattern matching and substitution are well typed [22].

The β and η rules follow the standard rules from the λ-calculus, where # is a *freshness* predicate, denoting that variable x does not appear inside term t.

For recursive **letrec** bindings, the β_{letrec} rule substitutes any occurrence of the bound variable x in t_2 with **letrec** $x = t_1$ **in** t_1, ensuring that recursive uses of x inside t_1 can be substituted with t_1 through subsequent β_{letrec} reduction. The LetRecDistrib rule allows distributivity of functions over **letrec** expressions, stating that if a function f can be applied to the entire **letrec** expression, then this is equivalent to applying f to just the body term t_2.

Term elimination is via **case**, requiring rules for both β- and η-equality on case expressions, as well as rules for associativity and distributivity. In β_{case}, a term t is matched against a pattern p_j in the context of the term t_j through the use of the partial function $(t \rhd p_j)t_j = t'$ which may substitute terms bound in p_j into t_j to yield t' if the match is successful. This partial function is defined inductively:

$$\frac{}{(t \rhd \_)t' = t'}\ \rhd_{-} \qquad \frac{}{(t \rhd x)t' = [t/x]t'}\ \rhd_{var} \qquad \frac{(t \rhd p)t' = t''}{([t] \rhd [p])t' = t''}\ \rhd_{\Box} \qquad \frac{(t_i \rhd p_i)t'_i = t'_{i+1}}{(Ct_1,..,t_n \rhd C\,p_1 \ldots p_n)t'_1 = t'_{n+1}}\ \rhd_{C}$$

As a predicate to the β_{case} rule, we require that j be minimal, i.e. the first pattern p_j in $p_1 \ldots p_n$ for which $(t \rhd p_j)t_j = t'$ is defined. Rule η_{case} states that if all branches of the case expression share a common term t_2 which differs between branches only in the occurrences of terms that match the pattern used, then we can substitute t_1 for the pattern inside t_2.

$$
\begin{array}{rcll}
(\lambda x.t_2)\,t_1 & \equiv & t_2[t_1/x] & \beta \\
\lambda x.t\,x & \equiv & t & (x\#t)\ \eta \\
\\
\textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2 & \equiv & t_2[\textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_1/x] & \beta_{letrec} \\
f\,(\textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2) & \equiv & \textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ (f\,t_2) & \textsc{LetRecDsitrib} \\
\\
\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i} & \equiv & (t \rhd p_j)t_j & (\mathsf{minimal}(j))\ \beta_{case} \\
\textbf{case}\ t_1\ \textbf{of}\ \overline{p_i \to t_2[p_i/z]} & \equiv & t_2[t_1/z] & \eta_{case} \\
\textbf{case}\ (\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i})\ \textbf{of}\ \overline{p'_i \to t'_i} & \equiv & \textbf{case}\ t\ \textbf{of}\ \overline{p_i \to (\textbf{case}\ t_i\ \textbf{of}\ \overline{p'_i \to t'_i})} & \textsc{CaseAssoc} \\
f\,(\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i}) & \equiv & \textbf{case}\ t\ \textbf{of}\ \overline{p_i \to (f\,t_i)} & \textsc{CaseDistrib} \\
\textbf{case}\ [\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i}]\ \textbf{of}\ \overline{[p'_i] \to t'_i} & \equiv & \textbf{case}\ [t]\ \textbf{of}\ \overline{[p_i] \to \textbf{case}\ [t_i]\ \textbf{of}\ \overline{[p'_i] \to t'_i}} & (\mathsf{lin}(p_i))\ [\textsc{CaseAssoc}]
\end{array}
$$

$$
\frac{\Gamma \vdash t : \Box_r A \qquad r \vdash p_i : A \rhd \Delta_i \qquad \Delta_i \vdash p_i : A \qquad 1 \sqsubseteq r}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ \overline{[p_i] \to p_i} \equiv \textbf{case}\ t\ \textbf{of}\ [x] \to x : A}\ \textsc{CaseGen}
$$

Figure 3: Equational theory for GrMini^P

Associativity of case expressions is provided by the CaseAssoc rule. This rule allows us to restructure nested case expressions such that the output terms t_i of the inner case may be matched against the patterns of the outer case, to achieve the same resulting output terms t'_i. The [CaseAssoc] rule provides a graded alternative to the CaseAssoc rule, where the nested case expression is graded, provided that the patterns p'_i of the outer case expression are also graded. Notably, this rule only holds when the patterns of the inner case expression are linear (i.e., variable or constant) so that there are no nested box patterns, represented via the $\mathsf{lin}(p_i)$ predicate. As with **letrec**, distributivity of functions over a case expression is given by CaseDistrib.

Lastly generalisation of an arbitrary boxed pattern to a variable is permitted through the CaseGen rule. Here, a boxed pattern $[p_i]$ and the output term of the case may be converted to a variable if the output term is equivalent to the pattern inside the box. The term t being matched against must therefore have a grade approximatable by 1, as witnessed by the predicate $1 \sqsubseteq r$ in the typing derivation.

3 Automatically Deriving *push* and *pull*

Now that we have established the core theory, we describe the algorithmic calculation of distributive laws in GrMini^P. Note that whilst GrMini^P is simply typed (monomorphic), it includes type variables (ranged over by α) to enable the distributive laws to be derived on parametric types. In the implementation, these will really be polymorphic type variables, but the derivation procedure need only treat them as some additional syntactic type construct.

Notation Let $\mathsf{F} : \mathsf{Type}^n \to \mathsf{Type}$ be an n-ary type constructor (i.e. a constructor which takes n type arguments), whose free type variables provide the n parameter types. We write $\mathsf{F}\overline{\alpha_i}$ for the application of F to type variables α_i for all $1 \leq i \leq n$.

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

$$[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \mathsf{F}(\overline{\Box_r \alpha_i})$$

where we require $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$ due to the [PCON] rule (e.g., if F contains a sum).

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{push}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{push}}\ z$ given by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{push}}$ where Σ is a context of *push* combinators for the recursive type variables:

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

In the case of *push* on a value of type $\mathbf{1}$, we pattern match on the value, eliminating the graded modality via the unboxing pattern match and returning the unit value. For type variables, *push* is simply the identity of the value, while for recursion variables we lookup the X's binding in Σ and apply it to the value. For sum and product types, *push* works by pattern matching on the type's constructor(s) and then inductively applying *push* to the boxed arguments, re-applying them to the constructor(s). Unlike *pull* below, the *push* operation can be derived for function types, with a contravariant use of *pull*. For recursive types, we inductively apply *push* to the value with a fresh recursion variable bound in Σ, representing a recursive application of push.

There is no derivation of a distributive law for types which are themselves graded modalities (see further work discussion in Section 8).

The appendix [17] proves that $[\![A]\!]_{\mathsf{push}}$ is type sound, i.e., its derivations are well-typed.

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

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

Type constructor F here is applied to n arguments each of the form $\Box_{r_i}\alpha_i$, i.e., each with a different grading of which the greatest-lower bound[4] $\bigwedge_{i=1}^{n} r_i$ is the resulting grade (see `pullPair` from Section 1.1).

[4]The greatest-lower bound $\wedge$ is partial operation which can be defined in terms of the semiring's pre-order: $r \wedge s = t$ if $t \sqsubseteq r$, $t \sqsubseteq s$ and there exists no other t' where $t' \sqsubseteq r$ and $t' \sqsubseteq s$ and $t \sqsubseteq t'$.

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{pull}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{pull}}\ z$ where:

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

Just like *push*, we cannot apply *pull* to graded modalities themselves. Unlike *push*, we cannot apply *pull* to function types. That is, we cannot derive a distributive law of the form $(\Box_r A \multimap \Box_r B) \multimap \Box_r(A \multimap B)$ since introducing the concluding $\Box_r$ would require the incoming function $(\Box_r A \multimap \Box_r B)$ to itself be inside $\Box_r$ due to the promotion rule (PR), which does not match the type scheme for *pull*.

The rest of the derivation above is similar but dual to that of *push*.

The appendix [17] proves that $[\![A]\!]_{\mathsf{pull}}$ is type sound, i.e., its derivations are well-typed.

Example 3.1. To illustrate the above procedures, the derivation of $[\![(\alpha \otimes \alpha) \multimap \beta]\!]_{\mathsf{push}}$ is:

$$
\begin{aligned}
&\lambda z.[\![(\alpha \otimes \alpha) \multimap \beta]\!]^{\emptyset}_{\mathsf{push}}\ z : \Box_r((\alpha \otimes \alpha) \multimap \beta) \multimap ((\Box_r\alpha \otimes \Box_r\alpha) \multimap \Box_r\beta) \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ [\![\alpha \otimes \alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y\ \mathbf{of}\ [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x',y') \to \\
&\qquad\qquad\qquad\qquad \mathbf{case}\ ([\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ x', [\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y')\ \mathbf{of}\ ([u],[v]) \to [(u,v)])\ \mathbf{of}\ [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x',y') \to \mathbf{case}\ (x',y')\ \mathbf{of}\ ([u],[v]) \to [(u,v)])\ \mathbf{of}\ [u] \to [(f\ u)]
\end{aligned}
$$

Remark 1. One might ponder whether linear logic's exponential $!A$ [12] is modelled by the graded necessity modality over $\mathbb{N}_\infty$ intervals, i.e., with $!A \triangleq \Box_{0..\infty}A$. This is a reasonable assumption, but $\Box_{0..\infty}A$ has a slightly different meaning to $!A$, exposed here: whilst $[\![A \otimes B]\!]_{\mathsf{push}} : \Box_{0..\infty}(A \otimes B) \multimap (\Box_{0..\infty}A \otimes \Box_{0..\infty}B)$ is derivable in GRMINI^P, linear logic does not permit $!(A \otimes B) \multimap (!A \otimes !B)$. Models of ! provide only a monoidal functor structure which gives *pull* for $\otimes$, but not *push* [2]. This structure can be recovered in Granule through the introduction of a partial type-level operation which selectively disallows *push* for $\otimes$ in semirings which model the ! modality of linear logic[5] [15].

The algorithmic definitions of 'push' and 'pull' can be leveraged in a programming context to automatically yield these combinators for practical purposes. We discuss how this is leveraged inside the Granule compiler in Section 4 and two techniques for leveraging it for Linear Haskell in Section 5. Before that, we study the algebraic behaviour of the derived distributive laws.

3.1 Properties

We consider here the properties of these derived operations. Prima facie, the above *push* and *pull* operations are simply distributive laws between two (parametric) type constructors F and $\Box_r$, the latter being the graded modality. However, both F and $\Box_r$ have additional structure. If the mathematical terminology

[5] The work in [15] arose as a result of the writing of this paper.

of 'distributive laws' is warranted, then such additional structure should be preserved by *push* and *pull* (e.g., as in how a distributive law between a monad and a comonad must preserve the behaviour of the monad and comonad operations after applying the distributive law [25]); we explain here the relevant additional structure and verify the distributive law properties.

Firstly, we note that these distributive laws are mutually inverse:

Proposition 3.1 (Pull is right inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i\in[1..n]}$ *and for all grades* $r \in \mathscr{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}([\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}) = id\ : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \Box_r \mathsf{F}\overline{\alpha_i}$$

Proposition 3.2 (Pull is left inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i\in[1..n]}$ *and for all grades* $r \in \mathscr{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}([\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}) = id\ : \mathsf{F}(\Box_r\overline{\alpha_i}) \multimap \mathsf{F}(\Box_r\overline{\alpha_i})$$

The appendix [17] gives the proofs, leveraging the equational theory of GrMini^P.

Applying a mathematical perspective, $\Box_r$ is also an endofunctor with its object mapping provided by the type constructor itself and its morphism mapping behaviour defined as follows:

Definition 3.1 ($\Box_r$ functor)**.** Given a function $f : \alpha \multimap \beta$ (a closed function term) then $\Box_r f : \Box_r\alpha \multimap \Box_r\beta$ is the morphism mapping of the endofunctor $\Box_r$ defined:

$$\Box_r\ f = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ [y] \to [f\ y]$$

For types $\mathsf{F}\alpha$ we can also automatically derive the morphism mapping of a covariant functor, which we write as $[\![\mathsf{F}\alpha]\!]_{\mathsf{fmap}}$ whose definition is standard (e.g., applied in Haskell [21]) given in the appendix [17]. Distributive laws between endofunctors should be natural transformations, which is indeed the case for our derivations:

Proposition 3.3 (Naturality of push)**.** *For all unary type constructors* F *such that* $[\![\mathsf{F}\alpha]\!]_{\mathsf{push}}$ *is defined, and given a closed function term* $f : \alpha \multimap \beta$*, then:* $[\![\mathsf{F}]\!]_{\mathsf{fmap}}\Box_r f \circ [\![\mathsf{F}\alpha]\!]_{\mathsf{push}} = [\![\mathsf{F}\beta]\!]_{\mathsf{push}} \circ \Box_r[\![\mathsf{F}]\!]_{\mathsf{fmap}} f$*, i.e.:*

$$\begin{array}{ccc} \alpha & & \Box_r\mathsf{F}\alpha & \xrightarrow{[\![\mathsf{F}\alpha]\!]_{\mathsf{push}}} & \mathsf{F}\Box_r\alpha \\ {\scriptstyle f}\downarrow & & {\scriptstyle \Box_r[\![\mathsf{F}]\!]_{\mathsf{fmap}}f}\downarrow & & \downarrow{\scriptstyle [\![\mathsf{F}]\!]_{\mathsf{fmap}}\Box_r f} \\ \beta & & \Box_r\mathsf{F}\beta & \xrightarrow[[\![\mathsf{F}\beta]\!]_{\mathsf{push}}]{} & \mathsf{F}\Box_r\beta \end{array}$$

Proposition 3.4 (Naturality of pull)**.** *For all unary type constructors* F *such that* $[\![\mathsf{F}\alpha]\!]_{\mathsf{pull}}$ *is defined, and given a closed function term* $f : \alpha \multimap \beta$*, then:* $\Box_r[\![\mathsf{F}]\!]_{\mathsf{fmap}} f \circ [\![\mathsf{F}\alpha]\!]_{\mathsf{pull}} = [\![\mathsf{F}\beta]\!]_{\mathsf{pull}} \circ [\![\mathsf{F}]\!]_{\mathsf{fmap}}\Box_r f$*, i.e.:*

$$\begin{array}{ccc} \alpha & & \mathsf{F}\Box_r\alpha & \xrightarrow{[\![\mathsf{F}\alpha]\!]_{\mathsf{pull}}} & \Box_r\mathsf{F}\alpha \\ {\scriptstyle f}\downarrow & & {\scriptstyle [\![\mathsf{F}]\!]_{\mathsf{fmap}}\Box_r f}\downarrow & & \downarrow{\scriptstyle \Box_r[\![\mathsf{F}]\!]_{\mathsf{fmap}} f} \\ \beta & & \mathsf{F}\Box_r\beta & \xrightarrow[[\![\mathsf{F}\beta]\!]_{\mathsf{pull}}]{} & \Box_r\mathsf{F}\beta \end{array}$$

The appendix [17] gives the proofs. Note that the naturality results here are for cases of unary type constructors F that are covariant functors, written with a single parameter α. This can easily generalise to n-ary type constructors.

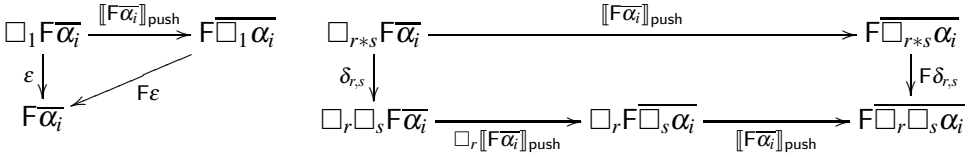
Not only is $\Box_r$ an endofunctor but it also has the structure of a *graded comonad* [10, 20, 23, 24].

Definition 3.2 (Graded comonadic operations)**.** The GRMINI^P calculus (and Granule) permits the derivation of graded comonadic operations [22] for the semiring graded necessity $\Box_r$, defined:

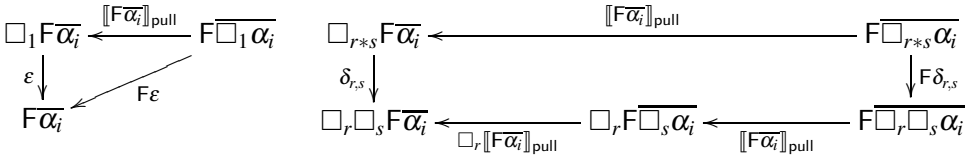
$$\varepsilon_A : \Box_1 A \multimap A = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ [z] \to z$$
$$\delta_{r,s,A} : \Box_{r*s} A \multimap \Box_r \Box_s A = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ [z] \to [[z]]$$

The derived distributive laws preserve these graded comonadic operations i.e., the distributive laws are well-behaved with respect to the graded comonadic structure of $\Box_r$, captured by the following properties:

Proposition 3.5 (Push preserves graded comonads)**.** *For all* F *such that* $[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{push}}$ *is defined and* F *does not contain* $\multimap$ *(to avoid issues of contravariance in* F*) then:*



Proposition 3.6 (Pull preserves graded comonads)**.** *For all* F *such that* $[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{pull}}$ *is defined then:*



The appendix [17] gives the proofs.

4 Implementation in Granule

The Granule type checker implements the algorithmic derivation of *push* and *pull* distributive laws as covered in the previous section. Whilst the syntax of GRMINI^P types had unit, sum, and product types as primitives, in Granule these are provided by a more general notion of type constructor which can be extended by user-defined, generalized algebraic data types (GADTs). The procedure outlined in Section 3 is therefore generalised slightly so that it can be applied to any data type: the case for $A \oplus B$ is generalised to types with an arbitrary number of data constructors.

Our deriving mechanism is exposed to programmers via explicit (visible) type application (akin to that provided in GHC Haskell [8]) on reserved names `push` and `pull`. Written `push @T` or `pull @T`, this signals to the compiler that we wish to derive the corresponding distributive laws at the type `T`. For example, for the `List : Type → Type` data type from the standard library, we can write the expression `push @List` which the type checker recognises as a function of type:

```
push @List : ∀ {a : Type, s : Semiring, r : s} . {1 ⩽ r} ⇒ (List a) [r] → List (a [r])
```

Note this function is not only polymorphic in the grade, but polymorphic in the semiring itself. Granule identifies different graded modalities by their semirings, and thus this operation is polymorphic in the graded modality. When the type checker encounters such a type application, it triggers the derivation procedure of Section 3, which also calculates the type. The result is then stored in the state of the frontend to be passed to the interpreter (or compiler) after type checking. The derived operations are memoized so that they need not be re-calculated if a particular distributive law is required more than once. Otherwise, the implementation largely follows Section 3 without surprises, apart from some additional machinery for specialising the types of data constructors coming from (generalized) algebraic data types.

Examples Section 1 motivated the crux of this paper with a concrete example, which we can replay here in concrete Granule, using its type application technique for triggering the automatic derivation of the distributive laws. Previously, we defined `pushPair` by hand which can now be replaced with:

```
push @(,) : ∀ {a, b : Type, s : Semiring, r : s} . (a, b) [r] → (a [r], b [r])
```

Note that in Granule `(,)` is an infix type constructor for products as well as terms. We could then replace the previous definition of `fst'` from Section 1 with:

```
fst' : ∀ {a, b : Type, r : Semiring} . {0 ⩽ r} ⇒ (a, b) [r] → a
fst' = let [x'] = fst (push @(,) x) in x'
```

The point however in the example is that we need not even define this intermediate combinator, but can instead write the following wherever we need to compute the first projection of `myPair : (a, b) [r]`:

```
extract (fst (push @(,) myPair)) : a
```

We already saw that we can then generalise this by applying this first projection inside of the list `myPairList : (List (a, b)) [r]` directly, using `push @List`.

In a slightly more elaborate example, we can use the `pull` combinator for pairs to implement a function that duplicates a pair (given that both elements can be consumed twice):

```
copyPair : ∀ {a, b : Type} . (a [0..2], b [2..4]) → ((a, b), (a, b))
copyPair x = copy (pull @(,) x) -- where, copy : a [2] → (a, a)
```

Note `pull` here computes the greated-lower bound of intervals `0..2` and `2..4` which is `2..2`, i.e., we can provide a pair of `a` and `b` values which can each be used exactly twice, which is what is required for `copy`.

As another example, interacting with Granule's indexed types (GADTs), consider a simple programming task of taking the head of a sized-list (vector) and duplicating it into a pair. The `head` operation is typed: `head : ∀ {a : Type, n : Nat} . (Vec (n + 1) a) [0..1] → a` which has a graded modal input 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, e.g., 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 usable twice
```

Here we "unbox" the graded modal value of type `a [2]` to get a non-linear variable `y` which we can use precisely 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 *push* on the inner graded modality `[2]` to get `(Vec (n + 1) (a [2])) [0..1]`.

Using `push @Vec` 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 = copy . head . boxmap [push @Vec] . disject
```

which employs combinators from the standard library and the derived distributive law, of type:

```
boxmap    : ∀ {a b : Type, s : Semiring, r : s}        . (a  → b) [r] → a [r] → b [r]
disject   : ∀ {a : Type, s : Semiring, n m : s}        . a [m * n] → (a [n]) [m]
push @Vec : ∀ {a : Type, n : Nat, s : Semiring, r : s} . (Vec n a) [r] → Vec n (a [r])
```

5 Application to Linear Haskell

While Granule has been pushing the state-of-the-art in graded modal types, similar features have been added to more mainstream languages. Haskell has recently added support for linear types via an underlying graded system which enables linear types as a smooth extension to GHC's current type system [3].[6] Functions may be linear with respect to their input (meaning, the function will consume its argument exactly once if the result is consumed exactly once), but they can also consume their argument `r` times for some multiplicity `r` via explicit 'multiplicity polymorphism'. Unlike Granule, Linear Haskell limits this multiplicity to the set of either one or many (the paper speculates about extending this with zero)—full natural numbers or other semirings are not supported.

In Linear Haskell, the function type (`a %r -> b`) can be read as "a function from `a` to `b` which uses `a` according to `r`" where `r` is either 1 (also written as `'One`) or ω (written as `'Many`). For example, the following defines and types the linear and non-linear functions `swap` and `copy` in Linear Haskell:

```
1 {-# LANGUAGE LinearTypes #-}
2 import GHC.Types
3
4 swap :: (a %1 -> b %1 -> c) %1 -> (b %1 -> a %1 -> c)
5 swap f x y = f y x
6
7 copy :: a %'Many -> (a, a)
8 copy x = (x, x)
```

Assigning the type `a %1 -> (a, a)` to `copy` would result in a type error due to a mismatch in multiplicity.

The approach of Linear Haskell (as formally defined by Bernardy et al. [3]) resembles earlier coeffect systems [11, 23] and more recent work on graded systems which have appeared since [1, 7]. In these approaches there is no underlying linear type system (as there is in Granule) but grading is instead pervasive with function arrows carrying a grade describing the use of their parameter. Nevertheless, recent systems also provide a graded modality as this enables more fine-grained usage information to be ascribed to compound data. For example, without graded modalities it cannot be explained that the first projection on a pair uses the first component once and its second component not at all (instead a single grade would have to be assigned to the entire pair).

We can define a graded modality in Linear Haskell via the following `Box` data type that is parameterized over the multiplicity `r` and the value type `a`, defined as:

```
1 data Box r a where { Box :: a %r -> Box r a }
```

A `Box` type is necessary to make explicit the notion that a value may be consumed a certain number of times, where ordinarily Linear Haskell is concerned only with whether individual functions consume their arguments linearly or not. Thus, distributive laws of the form discussed in this paper then become useful in practice when working with Linear Haskell.

[6] Released as part of GHC v9.0.1 in February 2021 `https://www.haskell.org/ghc/download_ghc_9_0_1.html`

The `pushPair` and `pullPair` functions from Section 1 can then be implemented in Linear Haskell with the help of this `Box` type:

```
pushPair :: Box r (a, b) %1 -> (Box r a, Box r b)
pushPair (Box (x, y)) = (Box x, Box y)

pullPair :: (Box r a, Box r b) %1 -> Box r (a, b)
pullPair (Box x, Box y) = Box (x, y)
```

Interestingly, `pushPair` could also be implemented as a function of type `(a, b) %r -> (Box r a, Box r b)`, and in general we can formulate push combinators typed `(f a) %r -> f (Box r a)`, i.e., consuming the input `r` times, returning a box with multiplicity `r`, but we stick with the above formulation for consistency.

While more sophisticated methods are outlined in this paper for automatically synthesizing these functions in the context of Granule, in the context of Linear Haskell (which has a simpler notion of grading) distributive laws over unary and binary type constructors can be captured with type classes:

```
class Pushable f where
  push :: Box r (f a) %1-> f (Box r a)
class Pushable2 f where
  push2 :: Box r (f a b) %1-> f (Box r a) (Box r b)

class Pullable f where
   pull :: f (Box r a) %1-> Box r (f a)
class Pullable2 f where
  pull2 :: f (Box r a) (Box r b) %1-> Box r (f a b)
```

Separate classes here are defined for unary and binary cases, as working generically over both is tricky in Haskell. Implementing an instance of `Pushable2` for pairs is then:

```
instance Pushable2 (,) where
  push2 (Box (x, y)) = (Box x, Box y)
```

This implementation follows the procedure of Section 3. A pair type $A \otimes B$ is handled by pattern matching on `(Box (x, y))` and boxing both fields in the pair `(Box x, Box y)`.

A Haskell programmer may define instances of these type classes for their own data types, but as discussed in Section 1, this is tedious from a software engineering perspective. Template Haskell is a meta-programming system for Haskell that allows compile-time generation of code, and one of the use cases for Template Haskell is generating boilerplate code [26]. The instances of `Pushable` and `Pullable` for algebraic data types are relatively straightforward, so we implemented procedures that will automatically generate these type class instances for arbitrary user-defined types (though types with so-called "phantom" parameters are currently not supported).

For example, if a programmer wanted to define a `Pushable` instance for the data type of a linear `List a`, they would write:

```
data List a where
  Cons :: a %1-> List a %1-> List a
  Nil  :: List a
$(derivePushable ''List)
```

Here, `derivePushable` is a Template Haskell procedure[7] that takes a name of a user-defined data type and emits a top-level declaration, following the strategy outlined in Section 3. For the `List a` data type above, we can walk through the type as `derivePushable` would. Because `List a` has two constructors, a case statement is necessary (our code generator will use a 'case lambda'). This case will have branches

[7] Available online: `https://github.com/granule-project/deriving-distributed-linear-haskell`

for each of the `Cons` and `Nil` constructors—in the body of the former it must box the first field (of type `a`) and recursively apply `push` to the second field (of type `List a`) after boxing it, and in the body of the latter it must simply return `Nil`. The full code generated by `derivePushable ''List` is given below.

```
derivePushable ''List
======>
instance Pushable List where
  push
    = \case
        Box (Cons match_a4BD match_a4BE)
          -> (Cons (Box match_a4BD)) (push (Box match_a4BE))
        Box Nil -> Nil
```

Later, in Section 7, we will discuss other combinators and type classes that are useful in Linear Haskell.

The applicability of our proposed deriving method to Haskell shows that it is useful beyond the context of the Granule project. Despite Haskell not having a formal semantics, we believe the same equational reasoning can be applied to show that the properties in Section 3.1 also hold for these distributive laws in Linear Haskell. As more programming languages adopt type system features similar to Granule and Linear Haskell, we expect that deriving/synthesizing or otherwise generically implementing combinators derived from distributive laws will be increasingly useful.

6 Typed-analysis of Consumption in Pattern Matching

This paper's study of distributive laws provides an opportunity to consider design decisions for the *typed analysis of pattern matching* since the operations of Section 3 are derived by pattern matching in concert with grading. We compare here the choices made surrounding the typing of pattern matching in four works (1) Granule and its core calculus [22] (2) the graded modal calculus Λ^p of Abel and Bernardy [1] (3) the dependent graded system GRAD of Choudhury et al. [7] and (4) Linear Haskell [3].

Granule Pattern matching against a graded modality $\Box_r A$ (with pattern $[p]$) in Granule is provided by the PBOX rule (Figure 2) which triggers typing pattern p 'under' a grade r at type A. This was denoted via the optional grade information $r \vdash p : A \rhd \Gamma$ which then pushes grading down onto the variables bound within p. Furthermore, it is only under such a pattern that wildcard patterns are allowed ([PWILD]), requiring $0 \sqsubseteq r$, i.e., r can approximate 0 (where 0 denotes weakening). None of the other systems considered here have such a facility for weakening via pattern matching.

For a type A with more than one constructor ($|A| > 1$), 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 (i.e., pattern matching on the 'tag' of the data constructors). 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. The idea that unary data types do not incur consumption (since no information is conveyed by its constructor) is a refinement here to the original Granule paper as described by Orchard et al. [22], which for [PCON] had only the premise $1 \sqsubseteq r$ rather than $|A| > 1 \implies 1 \sqsubseteq r$ here (although the implementation already reflected this idea).

The Λ^p calculus Abel and Bernardy's unified modal system Λ^p is akin to Granule, but with pervasive grading (rather than base linearity) akin to the coeffect calculus [23] and Linear Haskell [3] (discussed

in Section 5). Similarly to the situation in Granule, Λ^p also places a grading requirement when pattern matching on a sum type, given by the following typing rule in their syntax [1, Fig 1, p.4]:

$$\frac{\gamma\Gamma \vdash t : A_1 + A_2 \qquad \delta\Gamma, x_i :^q A_i \vdash u_i : C \qquad q \leq 1}{(q\gamma + \delta)\Gamma \vdash \mathsf{case}^q\ t\ \mathsf{of}\ \{\mathsf{inj}_1 x_1 \mapsto u_1; \mathsf{inj}_2 x_2 \mapsto u_2\} : C} +\text{-ELIM}$$

The key aspects here are that variables x_i bound in the case are used with grade q as denoted by the graded assumption $x_i :^q A_i$ in the context of typing u_i and then that $q \leq 1$ which is exactly our constraint that $1 \sqsubseteq r$ (their ordering just runs in the opposite direction to ours). For the elimination of pair and unit types in Λ^p there is no such constraint, matching our idea that arity affects usage, captured in Granule by $|A| > 1 \implies 1 \sqsubseteq r$. Their typed-analysis of patterns is motivated by their parametricity theorems.

GRAD The dependent graded type system GRAD of Choudhury et al. also considers the question of how to give the typing of pattern matching on sum types, with a rule in their system [7, p.8] which closely resembles the $+$-ELIM rule for Λ^p:

$$\frac{\Delta; \Gamma_1 \vdash q : A_1 \oplus A_2 \qquad \Delta; \Gamma_2 \vdash b_1 : A_1 \xrightarrow{q} B \qquad \Delta; \Gamma_2 \vdash b_2 : A_2 \xrightarrow{q} B \qquad 1 \leq q}{\Delta; q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B} \text{ST}\textsc{case}$$

The direction of the preordering in GRAD is the same as that in Granule but, modulo this ordering and some slight restructuring, the case rule captures the same idea as Λ^p: "both branches of the base analysis *must* use the scrutinee at least once, as indicated by the $1 \leq q$ constraint." [7, p.8]. Choudhury et al., also provide a heap-based semantics which serves to connect the meaning of grades with a concrete operational model of usage, which then motivates the grading on sum elimination here. In the simply-typed version of GRAD, matching on the components of a product requires that each component is consumed linearly.

Linear Haskell The paper on Linear Haskell by Bernardy et al. [3] has a **case** expression for eliminating arbitrary data constructors, with grading similar to the rules seen above. Initially, this rule is for the setting of a semiring over $\mathscr{R} = \{1, \omega\}$ (described in Section 5) and has no requirements on the grading to represent the notion of inspection, consumption, or usage due to matching on (multiple) constructors. This is reflected in the current implementation where we can define the following sum elimination:

```
1 match :: (Either a b) %r -> (a %1 -> c) -> (b %1 -> c) -> c
2 match (Left x) f _  = f x
3 match (Right x) _ g = g x
```

However, later when considering the generalisation to other semirings they state that "*typing rules are mostly unchanged with the caveat that* case_π *must exclude* $\pi = 0$" [3, §7.2, p.25] where π is the grade of the **case** guard. This appears a more coarse-grained restriction than the other three systems, excluding even the possibility of Granule's weakening wildcard pattern which requires $0 \leq \pi$. Currently, such a pattern must be marked as `'Many` in Linear Haskell (i.e., it cannot explain that first projection on a pair does not use the pair's second component). Furthermore, the condition $\pi \neq 0$ does not require that π actually represents a consumption, unlike the approaches of the other three systems. The argument put forward by Abel and Bernardy for their restriction to mark a consumption ($q \leq 1$) for the sake of parametricity is a compelling one, and the concrete model of Choudhury et al. gives further confidence that this restriction captures well an operational model. Thus, it seems there is a chance for fertilisation between the works mentioned here and Linear Haskell's vital work, towards a future where grading is a key tool in the typed-functional programmer's toolbox.

7 Deriving Other Useful Structural Combinators

So far we have motivated the use of distributive laws, and demonstrated that they are useful in practice when programming in languages with linear and graded modal types. The same methodology we have been discussing can also be used to derive other useful generic combinators for programming with linear and graded modal types. In this section, we consider two structural combinators, `drop` and `copyShape`, in Granule as well as related type classes for dropping, copying, and moving resources in Linear Haskell.

7.1 A Combinator for Weakening ("drop")

The built-in type constants of Granule can be split into those which permit structural weakening C^{w} such as `Int`, `Char`, `String`, and those which do not $C^{!}$ such as `Handle` (file handles) and `Chan` (concurrent channels). Those that permit weakening contain non-abstract values that can in theory be systematically inspected in order to consume them. Granule provides a built-in implementation of `drop` for C^{w} types, which is then used by the following derivation procedure to derive weakening on compound types:

$$[\![A]\!]_{\mathsf{drop}} : A \multimap 1$$

for closed types A defined $[\![A]\!]_{\mathsf{drop}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{drop}} z$ by an intermediate derivation $[\![A]\!]^{\Sigma}_{\mathsf{drop}}$:

$$\begin{aligned}
[\![C^{w}]\!]^{\Sigma}_{\mathsf{drop}} z &= \texttt{drop}\ z \\
[\![1]\!]^{\Sigma}_{\mathsf{drop}} z &= \textbf{case}\ z\ \textbf{of}\ \mathsf{unit} \to \mathsf{unit} \\
[\![X]\!]^{\Sigma}_{\mathsf{drop}} z &= \Sigma(X) z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{drop}} z &= \textbf{case}\ z\ \textbf{of}\ \ \mathsf{inl}\ x \to [\![A]\!]_{\mathsf{drop}}(x);\ \ \mathsf{inr}\ y \to [\![B]\!]_{\mathsf{drop}}(y) \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{drop}} z &= \textbf{case}\ z\ \textbf{of}\ (x,y) \to \textbf{case}\ [\![A]\!]_{\mathsf{drop}}(x)\ \textbf{of}\ \mathsf{unit} \to \textbf{case}\ [\![B]\!]_{\mathsf{drop}}(y)\ \textbf{of}\ \mathsf{unit} \to \mathsf{unit} \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{drop}} z &= \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1}_{\mathsf{drop}}\ \textbf{in}\ f\ z
\end{aligned}$$

Note we cannot use this procedure in a polymorphic context (over type variables α) since type polymorphism ranges over all types, including those which cannot be dropped like $C^{!}$.

7.2 A Combinator for Copying "shape"

The "shape" of values for a parametric data types F can be determined by a function $\mathit{shape} : \mathsf{F}A \to \mathsf{F}1$, usually derived when F is a functor by mapping with $A \to 1$ (dropping elements) [18]. This provides a way of capturing the size, shape, and form of a data structure. Often when programming with data structures which must be used linearly, we may wish to reason about properties of the data structure (such as the length or "shape" of the structure) but we may not be able to drop the contained values. Instead, we wish to extract the shape but without consuming the original data structure itself.

This can be accomplished with a function which copies the data structure exactly, returning this duplicate along with a data structure of the same shape, but with the terminal nodes replaced with values of the unit type 1 (the 'spine'). For example, consider a pair of integers: `(1, 2)`. Then applying `copyShape` to this pair would yield `((), ()), (1, 2))`. The original input pair is duplicated and returned on the right of the pair, while the left value contains a pair with the same structure as the input, but with values replaced with `()`. This is useful, as it allows us to use the left value of the resulting pair to reason about the structure of the input (e.g., its depth / size), while preserving the original input. This is particularly useful for deriving size and length combinators for collection-like data structures.

As with "drop", we can derive such a function automatically:

$$[\![\mathsf{F}\alpha]\!]_{\mathsf{copyShape}} : \mathsf{F}\alpha \multimap \mathsf{F1} \otimes \mathsf{F}\alpha$$

defined by $[\![A]\!]_{\mathsf{copyShape}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{copyShape}} z$ by an intermediate derivation $[\![A]\!]^{\Sigma}_{\mathsf{copyShape}}$:

$$\begin{aligned}
[\![C^w]\!]^{\Sigma}_{\mathsf{copyShape}} z &= (\mathsf{unit},\ z)\\
[\![1]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of}\ \mathsf{unit} \to (\mathsf{unit},\mathsf{unit})\\
[\![\alpha]\!]^{\Sigma}_{\mathsf{copyShape}} z &= (\mathsf{unit}, z)\\
[\![X]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \Sigma(X) z\\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{copyShape}}(x)\ \mathbf{of}\ (s,\ x') \to (\mathsf{inl}\ s, \mathsf{inl}\ x');\\
&\qquad\qquad\qquad\qquad\ \ \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{copyShape}}(y)\ \mathbf{of}\ (s,\ y') \to (\mathsf{inr}\ s, \mathsf{inr}\ y')\\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of}\ (x,y) \to \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{copyShape}}(x)\ \mathbf{of}\ (s,\ x') \to\\
&\qquad\qquad \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{copyShape}}(y)\ \mathbf{of}\ (s',\ y') \to ((s,\ s'),\ (x',\ y'))\\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1 \otimes A}_{\mathsf{copyShape}}\ \mathbf{in}\ f\ z
\end{aligned}$$

The implementation recursively follows the structure of the type, replicating the constructors, reaching the crucial case where a polymorphically type $z : \alpha$ is mapped to (unit, z) in the third equation.

Granule implements both these derived combinators in a similar way to *push/pull* providing `copyShape` and `drop` which can be derived for a type `T` via type application, e.g. `drop @T : T → ()` if it can be derived. Otherwise, the type checker produces an error, explaining why `drop` is not derivable at type `T`.

7.3 Other Combinators in Linear Haskell

As previously covered in Section 5, we demonstrated that the *push* and *pull* combinators derived from distributed laws are useful in Haskell with its linear types extension, and we demonstrated that they can be automatically derived using compile-time meta-programming with Template Haskell.

To the best of our knowledge, nothing comparable to the `Pushable` and `Pullable` type classes proposed here has been previously discussed in the literature on Linear Haskell. However, several other type classes have been proposed for inclusion in the standard library to deal with common use cases when programming with linear function types.[8] One of these classes, `Consumable`, roughly corresponds to the `drop` combinator above, while the other two, `Dupable` and `Movable`, are for when a programmer wants to allow a data type to be duplicated or moved in linear code.

```
1  class Consumable a where
2    consume :: a %1-> ()
3
4  class Consumable a => Dupable a where
5    dup2 :: a %1-> (a, a)
6
7  class Dupable a => Movable a where
8    move :: a %1-> Ur a
```

The `consume` function is a linear function from a value to unit, whereas `dup` is a linear function from a value to a pair of that same value. The `move` linear function maps `a` to `Ur a`, where `Ur` is the "unrestricted" modality. Thus, `move` can be used to implement both `consume` and `dup2`:

[8] See the `linear-base` library: `https://github.com/tweag/linear-base`.

```
case move x of {Ur _ -> ()}     -- consume x
case move x of {Ur x -> x}      -- x
case move x of {Ur x -> (x, x)} -- dup2 x
```

A ‘copy shape’ class may also be a useful addition to Linear Haskell in the future.

8 Discussion and Conclusion

Section 6 considered, in some detail, systems related to Granule and different typed analyses of pattern matching. Furthermore, in applying our approach to both Granule and Linear Haskell we have already provided some detailed comparison between the two. This section considers some wider related work alongside ideas for future work, then concludes the paper.

Generic Programming Methodology The deriving mechanism for Granule is based on the methodology of generic functional programming [14], where functions may be defined generically for all possible data types in the language; generic functions are defined inductively on the structure of the types. This technique has notably been used before in Haskell, where there has been a strong interest in deriving type class instances automatically. Particularly relevant to this paper is the work on generic deriving [21], which allows Haskell programmers to automatically derive arbitrary class instances using standard datatype-generic programming techniques as described above. In this paper we opted to rely on compile-time metaprogramming using Template Haskell [26] instead, but it is possible that some of the combinators we describe could be implemented using generic deriving as well, which is future work.

Non-graded Distributive Laws Distributive laws are standard components in abstract mathematics. Distributive laws between categorical structures used for modelling modalities (like monads and comonads) are well explored. For example, Brookes and Geva defined a categorical semantics using monads combined with comonads via a distributive law capturing both intensional and effectful aspects of a program [5]. Power and Watanabe study in detail different ways of combining comonads and monads via distributive laws [25]. Such distributive laws have been applied in the programming languages literature, e.g., for modelling streams of partial elements [29].

Graded Distributive Laws Gaboardi et al. define families of graded distributive laws for graded monads and comonads [10]. They include the ability to interact the grades, e.g., with operations such as $\Box_{\iota(r,f)}\Diamond_f A \to \Diamond_{\kappa(r,f)}\Box_r A$ between a graded comonad $\Box_r$ and graded monad $\Diamond_f$ where ι and κ capture information about the distributive law in the grades. In comparison, our distributive laws here are more prosaic since they involve only a graded comonad (semiring graded necessity) distributed over a functor and vice versa. That said, the scheme of Gaboardi et al. suggests that there might be interesting graded distributive laws between $\Box_r$ and the indexed types, for example, $\Box_r(\mathsf{Vec}\,n\,A) \to \mathsf{Vec}\,(r*n)\,(\Box_1 A)$ which internally replicates a vector. However, it is less clear how useful such combinators would be in general or how systematic their construction would be. In contrast, the distributive laws explained here appear frequently and have a straightforward uniform calculation.

We noted in Section 3 that neither of our distributive laws can be derived over graded modalities themselves, i.e., we cannot derive $\mathit{push} : \Box_r\Box_s A \to \Box_s\Box_r A$. Such an operation would itself be a distributive law between two graded modalities, which may have further semantic and analysis consequences beyond the normal derivations here for regular types. Exploring this is future work, for which the previous work on graded distributive laws can provide a useful scheme for considering the possibilities here.

Furthermore, Granule has both graded comonads and graded monads so there is scope for exploring possible graded distributive laws between these in the future following Gaboardi et al. [10].

In work that is somewhat adjacent to this paper, we define program synthesis procedures for graded linear types [16]. This program synthesis approach can synthesis *pull* distributive laws that are equivalent to the algorithmically derived operations of this paper. Interestingly the program synthesis approach cannot derive the *push* laws though as its core theory has a simplified notion of pattern typing that doesn't capture the full power of Granule's pattern matches as described in this paper.

Conclusions The slogan of graded types is to imbue types with information reflecting and capturing the underlying program semantics and structure. This provides a mechanism for fine-grained intensional reasoning about programs at the type level, advancing the power of type-based verification. Graded types are a burgeoning technique that is gaining traction in mainstream functional programming and is being explored from multiple different angles in the literature. The work described here addresses the practical aspects of applying these techniques in real-world programming. Our hope is that this aids the development of the next generation of programming languages with rich type systems for high-assurance programming.

Acknowledgments Thanks to Harley Eades III and Daniel Marshall for discussion and comments on a draft and to the reviewers and participants at TLLA-Linearity 2020. This work was supported by the EPSRC, grant EP/T013516/1 *Verifying Resource-like Data Use in Programs via Types*. The first author is also supported by an EPSRC DTA.

References

[1] Andreas Abel & Jean-Philippe Bernardy (2020): *A Unified View of Modalities in Type Systems*. *Proc. ACM Program. Lang.* 4(ICFP), doi:10.1145/3408972.

[2] Nick Benton, Gavin Bierman, Valeria De Paiva & Martin Hyland (1992): *Linear lambda-calculus and categorical models revisited*. In: *International Workshop on Computer Science Logic*, Springer, pp. 61–84, doi:10.1007/3-540-56992-8_6.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2017): *Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language*. *Proc. ACM Program. Lang.* 2(POPL), doi:10.1145/3158093.

[4] Edwin Brady (2021): *Idris 2: Quantitative Type Theory in Practice*. In Anders Møller & Manu Sridharan, editors: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 194, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 9:1–9:26, doi:10.4230/LIPIcs.ECOOP.2021.9. Available at https://drops.dagstuhl.de/opus/volltexte/2021/14052.

[5] Stephen Brookes & Kathryn V Stone (1993): *Monads and Comonads in Intensional Semantics*. Technical Report, Pittsburgh, PA, USA. Available at https://dl.acm.org/doi/10.5555/865105.

[6] Aloïs Brunel, Marco Gaboardi, Damiano Mazza & Steve Zdancewic (2014): *A core quantitative coeffect calculus*. In: *European Symposium on Programming Languages and Systems*, Springer, pp. 351–370, doi:10.1007/978-3-642-54833-8_19.

[7] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg & Stephanie Weirich (2021): *A Graded Dependent Type System with a Usage-Aware Semantics*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1145/3434331.

[8] Richard A Eisenberg, Stephanie Weirich & Hamidhasan G Ahmed (2016): *Visible type application*. In: *European Symposium on Programming*, Springer, pp. 229–254, doi:10.1007/978-3-662-49498-1_10.

[9] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan & 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*, pp. 357–370, doi:10.1145/2429069.2429113.

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

[11] Dan R. Ghica & Alex I. Smith (2014): *Bounded linear types in a resource semiring*. In: *Programming Languages and Systems*, Springer, pp. 331–350, doi:10.1007/978-3-642-54833-8_18.

[12] Jean-Yves Girard (1987): *Linear logic*. *Theoretical computer science* 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[13] Jean-Yves Girard, Andre Scedrov & Philip J Scott (1992): *Bounded linear logic: a modular approach to polynomial-time computability*. *Theoretical computer science* 97(1), pp. 1–66, doi:10.1016/0304-3975(92)90386-T.

[14] Ralf Hinze (2000): *A new approach to generic functional programming*. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 119–132, doi:10.1145/325694.325709.

[15] Jack Hughes, Daniel Marshall, James Wood & Dominic Orchard (2021): *Linear Exponentials as Graded Modal Types*. In: *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*, Rome (virtual), Italy. Available at https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.

[16] Jack Hughes & Dominic Orchard (2020): *Resourceful Program Synthesis from Graded Linear Types*. In: *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*, pp. 151–170, doi:10.1007/978-3-030-68446-4_8.

[17] Jack Hughes, Michael Vollmer & Dominic Orchard (2021): *Deriving Distributive Laws for Graded Linear Types (Additional Material)*, doi:10.5281/zenodo.5575771. Available at https://doi.org/10.5281/zenodo.5575771.

[18] C Barry Jay & J Robin B Cockett (1994): *Shapely types and shape polymorphism*. In: *European Symposium on Programming*, Springer, pp. 302–316, doi:10.1007/3-540-57880-3_20.

[19] Shin-ya Katsumata (2014): *Parametric effect monads and semantics of effect systems*. In Suresh Jagannathan & Peter Sewell, editors: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ACM, pp. 633–646, doi:10.1145/2535838.2535846.

[20] Shin-ya Katsumata (2018): *A Double Category Theoretic Analysis of Graded Linear Exponential Comonads*. In Christel Baier & Ugo Dal Lago, editors: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, *Lecture Notes in Computer Science* 10803, Springer, pp. 110–127, doi:10.1007/978-3-319-89366-2_6.

[21] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring & Andres Löh (2010): *A Generic Deriving Mechanism for Haskell*. *SIGPLAN Not.* 45(11), p. 37–48, doi:10.1145/2088456.1863529.

[22] Dominic Orchard, Vilem-Benjamin Liepelt & Harley Eades III (2019): *Quantitative program reasoning with graded modal types*. *PACMPL* 3(ICFP), pp. 110:1–110:30, doi:10.1145/3341714.

[23] Tomas Petricek, Dominic Orchard & Alan Mycroft (2014): *Coeffects: a calculus of context-dependent computation*. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, ACM, pp. 123–135, doi:10.1145/2692915.2628160.

[24] Tomas Petricek, Dominic A. Orchard & Alan Mycroft (2013): *Coeffects: Unified Static Analysis of Context-Dependence*. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska & David Peleg, editors: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, *Lecture Notes in Computer Science* 7966, Springer, pp. 385–397, doi:10.1007/978-3-642-39212-2_35.

[25] John Power & Hiroshi Watanabe (2002): *Combining a monad and a comonad*. *Theoretical Computer Science* 280(1-2), pp. 137–162, doi:10.1016/S0304-3975(01)00024-X.

[26] Tim Sheard & Simon Peyton Jones (2002): *Template Meta-Programming for Haskell*. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, Association for Computing Machinery, New York, NY, USA, p. 1–16, doi:10.1145/581690.581691.

[27] Ross Street (1972): *The formal theory of monads*. *Journal of Pure and Applied Algebra* 2(2), pp. 149–168, doi:10.1016/0022-4049(72)90019-9.

[28] K. Terui (2001): *Light Affine Lambda Calculus and Polytime Strong Normalization*. In: LICS '01, IEEE Computer Society, pp. 209–220, doi:10.1109/LICS.2001.932498.

[29] Tarmo Uustalu & Varmo Vene (November 2006): *The Essence of Dataflow Programming*. *Lecture Notes in Computer Science* 4164, pp. 135–167, doi:10.1007/11894100_5.